Konzepte für eine sichere Schlüsselverwaltung

Thilo Planz [1]

Juni 2002

Diplomarbeit
am Lehrstuhl für Theoretische Informatik
der Technischen Universität Darmstadt

Prof. Dr. J. Buchmann

Betreuer: Alexander Wiesmaier

[1]TU Darmstadt, Fachbereich Informatik, Matrikelnr. 547479, mailto:planz@epost.de

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 5. Juni 2002

Thilo Planz
planz@epost.de

Vorwort

Die vorliegende Arbeit ist Teil des umfangreichen FlexiTrust-Projekts, welches die Entwicklung einer leistungsfähigen Trustcenter-Software zum Ziel hat. Meine Arbeit entstand aus dem Wunsch heraus, diese Software um Möglichkeiten der sicheren Verwaltung des Signaturschlüssels der Zertifizierungsinstanz sowie der sicheren Verwahrung von Benutzerschlüsseln zu ergänzen. Auch wenn ich mich nicht auf diese beiden Problemstellungen beschränkt habe, habe ich mich bemüht, sie nicht aus den Augen zu verlieren.

Das vorliegende Dokument ist nur ein Teil meiner Diplomarbeit, die zum großen Teil auch aus der konkreten Umsetzung der hier geschilderten Verfahren in Form einer Programmierarbeit bestand. Diese Implementierung wird im folgenden in ihren wesentlichen Zügen geschildert, für einen genauen Einblick sei auf die Dokumentation und den Quelltext im CVS-Repositorium verwiesen.

Besonderer Dank gilt meinem Betreuer Alexander Wiesmaier und Professor Johannes Buchmann, der für seine Studenten auch bei ungewöhnlichen Anliegen, so wie ich sie selbst in den letzten Monaten vorgebracht habe, stets ein offenes Ohr hat.

Inhaltsverzeichnis

1 Einleitung **5**

2 Secret-Sharing **7**
2.1 Einfaches Secret-Sharing 8
2.2 Redundantes Secret-Sharing 9
2.3 Verifizierbares Secret-Sharing 11
2.4 Weitergehende Eigenschaften von Secret-Sharing-Verfahren . . . 13

3 Key-Sharing **16**
3.1 Public-Key-Kryptographie 16
3.2 Modellierung von Key-Sharing 17
3.3 Das RSA-Verfahren 19
3.4 Einfaches RSA-Key-Sharing 21
3.5 Redundantes RSA-Key-Sharing 22
3.6 Verifizierbares RSA-Key-Sharing 27
3.7 Das ElGamal-Verfahren 29
3.8 ElGamal-Key-Sharing 31
3.9 verteilte Schlüsselerzeugung 32
3.10 Anwendungen 33

4 Schlüsselverwahrung **35**
4.1 Key Escrow 35
4.2 Schlüsselverwahrung 38
4.3 wiederherstellbare Schlüssel 39

5 Implementierung **44**
5.1 Kryptographie in Java 44
5.2 Überblick über die Java-Klassen 48
5.3 Programmierschnittstelle 65
5.4 Applikationen 71

6 Ausblick **76**

A Abkürzungsverzeichnis **77**

B Symbolverzeichnis **79**

C UML-Klassendiagramme **81**

D Benutzerhandbuch Kommandozeilenapplikationen **87**
D.1 FileSharer 87
D.2 KeyEscrow 88

E Benutzerhandbuch Programmierschnittstelle **89**
E.1 Secret-Sharing-Basisfunktionalität 89
E.2 Key-Sharing-Basisfunktionalität 89
E.3 Der Shamir-KeyStore 90
E.4 Der KeySharer-KeyStore 91
E.5 Das Key-Escrow-System 91

Index **91**

Kapitel 1

Einleitung

Durch die in den letzten Jahren rasant vorangeschrittene Digitalisierung von Informationen und deren Speicherung auf Rechnersystemen sowie durch die immer umfassendere Vernetzung dieser Rechnersysteme und die damit einhergehenden neuen Möglichkeiten zum Datenaustausch haben sich beträchtliche Probleme bezüglich der Datensicherheit ergeben.

Insbesondere besteht ein großer Widerspruch zwischen dem auf offenen Netzen basierenden Internet, das sich mittlerweile als absolut dominantes Kommunikationsmedium durchgesetzt hat und mit dessen Hilfe sich gewaltige Kostensenkungs- und Effizienzsteigerungen in fast allen rechnergestützten Abläufen erzielen lassen, und den Anforderungen zum Schutz von personenbezogenen oder unternehmenskritischen Daten.

In diesem Zusammenhang hat die mathematische Disziplin der Kryptographie, die sich mit dem Chiffrieren von Informationen beschäftigt, viel Aufmerksamkeit erhalten. Ihr ist es zu verdanken, daß mittlerweile zahlreiche Verfahren zur Verfügung stehen, mit denen Daten so stark verschlüsselt werden können, daß sie ein Unbefugter auch mit enormem Rechenaufwand nicht entschlüsselt kann, mit denen Dokumente vor unbemerkten Veränderungen geschützt werden können und mit denen man Nachrichten mit einer digitalen Signatur versehen kann, die eindeutig einer bestimmten Person zugeordnet werden kann.

Das wichtigste Problem beim Einsatz kryptographischer Verfahren besteht in der Verwaltung der dafür benötigten Schlüssel. In der traditionellen Kryptographie mußten sich der Sender und der Empfänger einer Nachricht vorab auf einen gemeinsamen Schlüssel verständigen und diesen fortan geheim halten. Um vertrauliche Nachrichten senden zu können mußte außerdem jeder Absender für jeden Empfänger einen anderen Schlüssel vorhalten. Wenn der geheime Schlüssel in die falschen Hände geriet, war die Sicherheit der Kommunikation verloren. Durch das Aufkommen der Public-Key-Kryptographie hat sich diese Lage deutlich verbessert: jetzt gab es separate Schlüssel für Sender und Empfänger und alle Sender konnten pro Empfänger denselben Schlüssel verwenden. Dieser Schlüssel mußte nicht mehr geheim gehalten werden, im Gegenteil sollte er möglichst allgemein bekannt gemacht werden. Das Problem den Empfängerschlüssel geheim zu halten blieb bestehen, wenn man ihn auch nicht mehr mit anderen (den Sendern) teilen mußte. Dazu kam das Problem, die

öffentlichen Schlüssel eindeutig und resistent gegen Betrugsversuche den teilnehmenden Personen zuzuordnen. Mit dieser Aufgabe wurden Zertifizierungsinstanzen betraut.

In dieser Arbeit, die Teil eines größeren Projektes zur Entwicklung einer Software zum Betrieb einer Zertifizierungsinstanz ist, werden algorithmische Verfahren zum Schutz der privaten Schlüssel vor Verlust und Spionage vorgestellt. Diese eignen sich insbesondere zum Schutz der wertvollen Schlüssel, die von einer solchen Instanz selbst verwaltet werden und an denen folglich die Sicherheit vieler anderer Schlüssel hängt. Die meisten der im folgenden geschilderten Algorithmen haben wir auch implementiert, so daß sie direkt verwendet werden können.

Kapitel 2

Secret-Sharing

Wenn ich mein Geheimnis verschweige, ist es mein Gefangener.
Lasse ich es entschlüpfen, bin ich sein Gefangener.
arabisches Sprichwort

Ein grundsätzliches Problem, das sich beim Einsatz eines kryptographischen Systems stellt, ist die sichere Verwahrung der geheimzuhaltenden Schlüssel. Während dem Betrieb hält ein solches System seine Schlüssel üblicherweise in einem geschützten Speicherbereich, dessen Inhalt bei einem Ausspähversuch oder sonstigem Angriff sofort unwiderbringlich gelöscht wird. Damit ist der Schlüssel dagegen geschützt, in unbefugte Hände zu fallen, aber durch seinen Verlust wird auch der weitere Betrieb des Systems unmöglich gemacht. Daher ist man gezwungen, eine oder mehrere nicht-flüchtige Kopien des Schlüssels an einem sicheren Ort zu hinterlegen. Mit jeder dieser Kopien steigt das Risiko, daß eine der Kopien von einem Angreifer erbeutet wird. Legt man allerdings zu wenige Kopien an, könnten alle gleichzeitig zerstört werden.

Eine Lösung des Problems haben unabhängig voneinander Adi Shamir [Sha79] und G. R. Blakley [Bla79] vorgeschlagen: Man erzeuge aus dem Geheimnis eine Anzahl neuer Geheimnisse, die für sich genommen keinen Rückschluß auf das ursprüngliche Geheimnis ermöglichen (Schutz gegen Spionage), aus denen dieses aber rekonstruiert werden kann, wenn nur genügend viele von ihnen vorliegen (Schutz gegen Verlust). Beispielsweise kann man mit den von Shamir und Blakley geschilderten Verfahren das Geheimnis auf fünf Teile aufteilen, von denen man später mindestens drei benötigt, um das Geheimnis wiederherzustellen.

Bevor wir das relativ bekannte Verfahren von Shamir in Abschnitt 2.2.1 erläutern, wollen wir in Abschnitt 2.1 eine einfachere Methode vorstellen, die jedoch nicht gegen den Verlust einzelner Anteile gesichert ist. Danach schildern wir in Abschnitt 2.3.1 eine Erweiterung des Shamir-Verfahrens. Zum Abschluß dieses Kapitels geben wir einen Überblick über weitergehende Anforderungen und Entwicklungen zum Thema Secret-Sharing. Eine spezielle Anforderung, nämlich die Verwendbarkeit geteilter Schlüssel ohne deren jemalige Rekonstruktion wird uns dann im nächsten Kapitel beschäftigen.

2.1 Einfaches Secret-Sharing

Eine sehr einfache Methode, eine geheime Information unter mehreren Teilnehmern zu verteilen, basiert auf der Idee der One-Time-Pad-Verschlüsselung. Hierbei wird das Geheimnis in eine Reihe von Summanden aufgeteilt und kann dann als Summe dieser Summanden wiederhergestellt werden. Die einzelnen Summanden sind rein zufällig und enthalten keinerlei Information über das Geheimnis. Das Verfahren läßt sich sehr effizient implementieren und ist beweisbar sicher. Allerdings werden sämtliche Teilnehmer benötigt, um das Geheimnis zu rekonstruieren.

2.1.1 One-Time-Pad-Verschlüsselung

Bei diesem Verfahren handelt es sich um ein Verschlüsselungsverfahren, das heißt es beschreibt, wie ein Sender eine Nachricht derart verschlüsseln kann, daß nur ein authorisierter Empfänger die Nachricht wieder entschlüsseln kann. Der Empfänger wird dadurch authorisiert, daß er sich mit dem Sender vorab auf einen gemeinsamen Schlüssel zur Ver- und Entschlüsselung der Nachricht geeinigt hat (man bezeichnet ein solches Verschlüsselungsverfahren als *symmetrisch*). Bei der Beschreibung des Verfahrens wollen wir davon ausgehen, daß es sich bei der Nachricht um eine Bitfolge handelt.

Um eine Nachricht zu verschlüsseln, müssen Sender und Empfänger vorab einen gemeinsamen Schlüssel vereinbart haben, bei dem es sich um eine zufällige Bitfolge handelt, die (mindestens) genauso lang ist wie die Nachricht. Jedes Bit der Nachricht wird dann mit dem entsprechenden Bit des Schlüssels (des One-Time-Pads) per XOR kombiniert. Beispielsweise würde aus dem Klartext `101011` unter Verwendung des Schlüssels `100100` der Schlüsseltext `001111`. Danach vernichtet der Sender seine Kopie des Schlüssels. Aufgrund der Eigenschaft, daß sich die zweifache Anwendung von XOR gegenseitig aufhebt ($a \oplus b \oplus b = a$) kann der Klartext durch erneute Kombination mit dem Schlüssel durch den Empfänger wieder sichtbar gemacht werden: `001111` $\oplus$ `100100` = `101011`. Für jeden anderen ist der Schlüsseltext nur eine rein zufällige Zeichenfolge.

Informationstheoretische Sicherheit Obwohl das One-Time-Pad-Verfahren sehr simpel aufgebaut ist, erfüllt es doch das stärkste bekannte Sicherheitskriterium, die informationstheoretische Sicherheit. Dies bedeutet, daß durch die Kenntnis des Schlüsseltextes (und sämtlicher öffentlicher Eigenschaften des Verfahrens) keinerlei Rückschlüsse auf den Klartext möglich sind, weil alle denkbaren Klartexte auch bei dem vorliegenden Schlüsseltext die gleiche Wahrscheinlichkeit aufweisen. Da dieses Forderung zuerst von Claude Shannon erhoben wurde, spricht man auch von Sicherheit nach Shannon. Shannon hat gezeigt, daß das One-Time-Pad-Verfahren diese Eigenschaft hat. Im Gegensatz dazu sind fast alle anderen Verfahren lediglich berechnungssicher (*computationally secure*), was bedeutet, daß ihre Sicherheit auf einem (hoffentlich) schwierig zu lösenden mathematischen Problem beruht.

Praktische Einsetzbarkeit Die Sicherheit des One-Time-Pad-Verfahrens beruht auf der Zufälligkeit des Schlüssels. Wenn er nicht wirklich zufällig erzeugt wurde, sondern durch einen Algorithmus (einen Pseudo-Zufallszahlengenerator), dann läßt sich die Shannon-Sicherheit nicht mehr beweisen. Darüber hinaus erfordert die Übermittlung des Schlüssels an den Empfänger ein vorheriges Treffen oder vertrauenswürdige Boten. Nicht zuletzt darf der Schlüssel nur ein einziges Mal benutzt werden. Damit ist der Einsatz des eigentlich simplen Verfahrens mit großem organisatorischen Aufwand verbunden (in der Tat wurde es vor allem für wichtige diplomatische und militärische Mitteilungen in der ersten Hälte des 20. Jahrhunderts verwendet, für die der hohe Aufwand zu rechtfertigen war).

2.1.2 Die XOR-Methode

Wenn der Absender einer mit dem One-Time-Pad verschlüsselten Nachricht den Schlüsseltext einem Boten übergibt, können wir uns dies als das Aufteilen der Nachricht auf zwei Personen vorstellen, den Boten und den Empfänger: Der Bote besitzt die verschlüsselte Nachricht und der Empfänger das zugehörige One-Time-Pad. Solange die beiden nicht zusammenarbeiten (was üblicherweise darin besteht, daß der Bote den Schlüsseltext an den Empfänger übergibt), kann keiner von beiden (und auch kein Dritter) die Nachricht rekonstruieren und verfügt nur über eine zufällige Bitfolge. Mit der in Abschnitt 2.2 eingeführten Sprechweise haben wir es hier mit einem $\binom{2}{2}$-Secret-Sharing-Verfahren zu tun.

Das Verfahren läßt sich leicht auf beliebig viele Teilnehmer verallgemeinern: Um eine geheime Information $s = s^{[m-1]}s^{[m-2]}\ldots s^{[1]}s^{[0]}$ mit m Bits auf t Teilnehmer zu verteilen erhalten alle Teilnehmer bis auf den letzten jeweils einen m-bit-langen Zufallsstring

$$s_i = s_i^{[m-1]}s_i^{[m-2]}\ldots s_i^{[1]}s_i^{[0]} \qquad s_i^{[j]} \in_R \{0,1\} \qquad 1 \leq i < t$$

und der letzte Teilnehmer erhält die XOR-Summe

$$s_t = s_t^{[m-1]}s_t^{[m-2]}\ldots s_t^{[1]}s_t^{[0]} \qquad s_t^{[j]} = s^{[j]} \oplus s_1^{[j]} \oplus \ldots \oplus s_{t-1}^{[j]}$$

aller anderen Anteile mit dem Geheimnis. Damit läßt sich (bei Vorliegen aller Teilgeheimnisse s_i) jedes Bit $s^{[j]}$ von s als $s^{[j]} = \bigoplus_i s_i^{[j]}$ zurückgewinnen.

2.2 Redundantes Secret-Sharing

Wie wir gesehen haben, ermöglicht es die XOR-Methode, ein Geheimnis auf mehrere Orte zu verteilen, wodurch es wesentlich besser dagegen geschützt ist, aufgedeckt zu werden. Auf diese Weise wird das Sicherheitsproblem einer replizierten Speicherung umgangen. Unglücklicherweise geht aber auch der Vorteil einer Replikation verloren, nämlich die Robustheit gegen den Verlust einzelner Anteile: Sobald auch nur einer der Anteile der XOR-Methode nicht mehr zur Verfügung steht, kann das Geheimnis nicht mehr rekonstruiert werden. Diesem Mangel treten redundante Secret-Sharing-Verfahren entgegen, bei denen bereits eine Teilmenge der Anteile genügt, um das Geheimnis zurückzugewinnen.

$\binom{t}{n}$-Secret-Sharing Die meisten Vertreter redundanter Secret-Sharing-Verfahren sind sogenannte Threshold-Verfahren (Schwellwertverfahren). Hierbei werden alle n ausgegebenen Anteile als gleichwertig betrachtet, so daß eine beliebige Kombination von mindestens t dieser Anteile ausreicht, um das Geheimnis zu rekonstruieren. Ein solches Verfahren wird dann als $\binom{t}{n}$-Secret-Sharing bezeichnet.

2.2.1 Das Shamir-Verfahren

Die bekannteste Umsetzung von redundantem Secret-Sharing ist das von Adi Shamir vorgeschlagene Polynominterpolationsverfahren [Sha79]. Um ein $\binom{t}{n}$-Secret-Sharing einer geheimen Zahl s zu betreiben, wählt der Geber ein zufälliges Polynom $f(x)$ von Grad $t-1$ mit Achsenabschnitt $f(0) = s$ und teilt jedem Teilnehmer i den Funktionswert $s_i = f(i)$ mit. Mit dem Satz von Lagrange kann dann aus t Funktionswerten das Polynom (inklusive des Achsenabschnittes s) rekonstruiert werden. Da das Polynom über einer primen Restklasse ausgewertet wird, sind bei Kenntnis von weniger als t Interpolationsstellen weiterhin alle denkbaren Achsenabschnitte möglich und gleichwahrscheinlich.

Erzeugung der Teilgeheimnisse Der Geber wählt zunächst eine Primzahl p, die größer ist als jede mögliche geheime Zahl s, und gibt sie öffentlich bekannt. Alle folgenden Berechnungen werden in der Restklasse $(\mathbb{Z}/p\mathbb{Z})$ ausgeführt. Da p eine Primzahl ist, sind insbesondere alle von Null verschiedenen Elemente der Restklasse invertierbar. Die Primzahl p muß nicht für jedes Geheimnis, das verteilt werden soll, neu gewählt werden, sondern kann auch als fester Bestandteil des Verfahrens angesehen und somit global bekannt sein.

Um die geheime Zahl $s \in (\mathbb{Z}/p\mathbb{Z})$ zu verteilen, wählt der Geber zufällige Koeffizienten $a_j \in_R (\mathbb{Z}/p\mathbb{Z}), 1 \leq j \leq t-1$, wodurch sich das Polynom

$$f(x) = s + a_1 x + \ldots + a_{t-1} x^{t-1} \pmod{p}$$

ergibt. Dieses Polynom hat den Grad $t-1$, wird also durch t seiner Stützstellen eindeutig festgelegt. Jeder Teilnehmer $i \in \{1 \ldots n\}$ erhält den Funktionswert $s_i = f(i)$ als seinen Anteil des Geheimnisses.

Rekonstruktion des Geheimnisses Wenn sich mindestens t der n Teilnehmer zusammenfinden, können sie das Geheimnis durch Offenlegung ihrer Teilgeheimnisse gemeinsam rekonstruieren. Wir wollen die Menge der Nummern dieser Teilnehmer als $\Lambda \subset \{1 \ldots n\}$ bezeichnen. Unter Verwendung der Interpolationsformel von Lagrange

$$f(x) = \sum_{i=0}^{t-1} a_i x^i = \sum_{i=1}^{t} f(x_i) \prod_{l=1, l \neq i}^{t} \frac{x_l - x}{x_l - x_i}$$

für beliebige Stützstellen x_i ergibt sich das Geheimnis s als

$$s = f(0) = \sum_{i \in \Lambda} s_i \prod_{l \in \Lambda \setminus \{i\}} \frac{l}{l-i} \pmod{p}.$$

Sicherheit des Verfahrens Ein Angreifer, dem es gelingt, bis zu $t-1$ Teilgeheimnisse s_i in Erkenntnis zu bringen, kann dadurch keine Rückschlüsse auf das Geheimnis s ziehen. Wenn $t-1$ Stützstellen des geheimen Polynoms bekannt sind, exisitiert für jeden denkbaren Wert s' genau ein Polynom, welches alle Stützstellen und diesen Wert s' beinhaltet. Aufgrund der Konstruktion des Polynoms ist jedes dieser möglichen Polynome gleichwahrscheinlich. Das Shamir-Verfahren ist somit informationstheoretisch sicher im Shannon'schen Sinne.

2.3 Verifizierbares Secret-Sharing

Eine Hauptmotivation für den Einsatz von Secret-Sharing ist die Möglichkeit, Angriffe auf die Vertraulichkeit und Verfügbarkeit des Geheimnisses abzuwehren. Durch das Shamir-Verfahren wird sichergestellt, daß das Geheimnis einem Angreifer nur in die Hände fallen kann, wenn er Kenntnis von einer Mindestmenge an Teilgeheimnissen erhält. Trotzdem kann ein Angreifer, der die Kontrolle über einige wenige Teilnehmer erhält, zumindest die Verfügbarkeit des Geheimnisses empfindlich stören, wenn er die Rekonstruktion sabotiert. Es ist nämlich bei der Kombination der Teilgeheimnisse nicht möglich zu erkennen, ob der Beitrag einzelner Teilnehmer korrekt war oder nicht. Das macht es schwierig, betrügerische Teilnehmer auszuschließen. Von einem robusten Secret-Sharing-Verfahren wird verlangt, daß es auch eine gewisse Menge an Teilnehmern verkraftet, die sich nicht an alle Einzelheiten des Protokolls halten. Eine wichtige Komponente robuster Secret-Sharing-Verfahren sind verifizierbare Teilgeheimnisse, die wir in diesem Abschnitt besprechen wollen.

Das Shamir-Verfahren funktioniert, wenn sich alle Teilnehmer (inklusive des Gebers) an die vorgeschriebenen Protokollabläufe halten. Das Geheimnis ist dann gegenüber Außenstehenden (und gegenüber einer Minderheit an Teilnehmern) geschützt, kann aber bei Bedarf durch Kooperation rekonstruiert werden. Man sagt, die Teilnehmer seien neugierig, aber ehrlich (*curious, but honest*). Teilnehmer, die sich nicht an das Protokoll halten, können die Rekonstruktion aufhalten. Außerdem kann ein Teilnehmer sich nicht sicher sein, daß sein Anteil überhaupt korrekt gebildet wurde (Übertragungsfehler, Sabotage, betrügerischer Geber). Beide Mängel können mit einem Verfahren von Torben Pedersen [Ped92] behoben werden, das auf einem Commitment des Gebers beruht und die Verifikation der Teilgeheimnisse ermöglicht. Dieses Verfahren zeichnet sich gegenüber anderen vorgeschlagenen Verfahren dadurch aus, daß es ohne Interaktionen zwischen den Teilnehmern oder mit dem Geber auskommt und daß durch die zusätzlichen Verifikationsinformationen keinerlei Hinweise auf das Geheimnis gegeben werden.

2.3.1 Das Pedersen-Verfahren

Das Verfahren benötigt eine weitere Primzahl q mit $q = mp + 1$ für ein beliebiges (möglicherweise kleines) m. Dadurch ergibt sich der Körper $(\mathbb{Z}/q\mathbb{Z})$, der über eine Untergruppe G_p der Ordnung p verfügt. Sei $g \in G_p$ ein erzeugendes Element von G_p und $h \in G_p$ ein weiteres Element aus G_p, für das der diskrete

Logarithmus $\log_g h$ bezüglich des Erzeugers g nicht bekannt ist. Der Geber kann nun ein Commitment für zwei Werte s und s' aus $(\mathbb{Z}/p\mathbb{Z})$eingehen, ohne damit Informationen über s oder s' preiszugeben:

$$E(s, s') = g^s h^{s'} \pmod{q}$$

Das Commitment an den (öffentlich gemachten) Wert $E(s, s')$ kann durch Offenlegung von s und s' überprüft werden. Es kann gezeigt werden, daß es unmöglich ist, das Commitment zu fälschen, ohne $\log_g h$ zu kennen [Ped92]. Dieses Commitment-Verfahren wird nun mit dem Shamir-Verfahren kombiniert. Dabei verteilt der Geber neben dem eigentlichen Geheimnis s eine zweite Zahl s' und gibt ein Commitment für die beiden dabei benutzten Polynome ab. Das Commitment kann dann von jedem Teilnehmer anhand seines Anteils (s_i, s'_i) überprüft werden.

Erzeugung der Teilgeheimnisse Um die geheime Zahl $s \in (\mathbb{Z}/p\mathbb{Z})$ zu verteilen, wählt der Geber zufällige Koeffizienten $a_j \in_R (\mathbb{Z}/p\mathbb{Z}), 1 \leq j \leq t-1$, wodurch sich das Polynom

$$f(x) = s + a_1 x + \ldots + a_{t-1} x^{t-1} \pmod{p}$$

ergibt. Zusätzlich verteilt der Geber die zufällige Zahl $s' \in (\mathbb{Z}/p\mathbb{Z})$ auf analoge Weise durch ein (vollständig zufälliges) Polynom

$$f'(x) = s' + a'_1 x + \ldots + a'_{t-1} x^{t-1} \pmod{p}.$$

Jeder Teilnehmer $i \in \{1 \ldots n\}$ erhält die Funktionswerte $s_i = f(i)$ und $s'_i = f'(i)$ als seinen Anteil des Geheimnisses. Außerdem gibt der Geber ein Commitment an die beiden Polynome ab, indem er

$$E_0 = E(s, s') = g^s h^{s'} \pmod{q}$$

und

$$E_j = E(a_j, a'_j) = g^{a_j} h^{a'_j} \pmod{q} \quad \text{für} 1 \leq j \leq t-1$$

veröffentlicht.

Verifikation eines Teilgeheimnisses Ein Teilgeheimnisses (s_i, s'_i) ist genau dann gültig, wenn die beiden Werte tatsächlich Stützstellen der vom Geber gewählten Polynome $f(x)$ und $f'(x)$ sind:

$$s_i = f(i) = s + \sum_{j=1}^{t-1} a_j i^j$$

und

$$s'_i = f'(i) = s' + \sum_{j=1}^{t-1} a'_j i^j.$$

Diese beiden Gleichungen können durch das Commitment des Gebers an die Koeffizienten überprüft werden, es muß nämlich für jeden Anteil (s_i, s'_i) gelten, daß

$$\prod_{j=0}^{t-1} E_j^{i^j} = (g^s h^{s'}) \prod_{j=1}^{t-1} (g^{a_j} h^{a'_j})^{i^j} = g^{s+\sum_{j=1}^{t-1} a_j i^j} h^{s'+\sum_{j=1}^{t-1} a'_j i^j} = g^{f(i)} h^{f'(i)}$$

$$= g^{s_i} h^{s'_i} \pmod{q}.$$

Rekonstruktion des Geheimnisses Die Rekonstruktion von s aus den s_i verläuft genau wie beim Shamir-Verfahren. Auf diese Weise könnten die Teilnehmer auch s' aus den s'_i berechnen, aber dies ist keine wertvolle Information.

Anmerkung zur Bedeutung von s' Das zweite „Geheimnis“ s' dient dazu, die informationstheoretische Sicherheit (nach Shannon) von s zu erhalten. Es wirkt dabei wie ein Blendfaktor. Wenn man diese Forderung nicht erhebt, kann man das ähnlichere, aber einfachere Verfahren von Feldman [Fel87] verwenden, in dem g^s bekannt wird. Damit ist s nur noch dadurch sicher, daß es schwierig ist, diskrete Logarithmen zu berechnen. Beispielsweise kann so auch das niederwertigste Bit von s nicht mehr geheim gehalten werden.

Die Verifizierbarkeit der Teilgeheimnisse hängt davon ab, daß der Geber bei seinem Commitment nicht betrügen kann. Dazu wäre er in der Lage, wenn er den diskreten Logarithmus $\log_g h$ berechnen könnte. In diesem Fall erlangt der Geber die Möglichkeit, Teilnehmern fehlerhafte Anteile unterzuschieben. Die Vertraulichkeit des Geheimnisses bleibt davon jedoch unbeeinflußt. Ein Teilnehmer (oder Außenstehender) kann durch Kenntnis von $\log_g h$ keine Rückschlüsse auf das Geheimnis ziehen, daß wie beim reinen Shamir-Verfahren informationstheoretisch sicher bleibt.

2.4 Weitergehende Eigenschaften von Secret-Sharing-Verfahren

Abschließend wollen wir noch einmal die bisher vorgestellten und weitere Eigenschaften von Secret-Sharing-Verfahren zusammenfassen und damit einen Überblick über die in der Literatur verwendete Terminologie sowie die verschiedenen Anforderungen an die Verfahren bieten.

- Ein Secret-Sharing-Verfahren heißt *perfekt*, wenn sich aus der Kenntnis einer für die Rekonstruktion unzureichender Menge an Teilgeheimnissen *keinerlei* Informationen über das Geheimnis ableiten lassen, die man nicht auch ohne Kenntnis eines einzigen Teilgeheimnisses hätte. Beim $\binom{t}{n}$-Shamir-Verfahren bedeutet dies, daß trotz $t-1$ bekannter Stützstellen weiterhin alle denkbaren Achsenabschnitte des geheimen Polynoms möglich und gleichwahrscheinlich sind.

- Ein Secret-Sharing-Verfahren heißt *redundant*, wenn nicht alle Teilgeheimnisse zur Rekonstruktion benötigt werden. Dadurch ist das Verfahren nicht anfällig gegen den Verlust einzelner Anteile.

- Wir sprechen von $\binom{t}{n}$-Secret-Sharing, wenn eine beliebige t-elementige Teilmenge der n ausgegebenen Anteile ausreicht, um das Geheimnis zu rekonstruieren. Damit sind automatisch alle Anteile gleichberechtigt. Im Gegensatz dazu gibt es Verfahren, die über unterschiedlich wertvolle Anteile verfügen oder explizite Gruppen (*access structures*) von Anteilen zur Rekonstruktion vorgeben.

- Offensichtlich führt ein fehlerhafter Beitrag eines Teilnehmers bei der Rekonstruktion zu einem fehlerhaften Ergebnis. Das richtige Ergebnis und die Identität des Teilnehmers läßt sich im allgemeinen dann nur durch versuchsweises Durchprobieren aller möglicher Gruppen von Teilnehmern errechnen. Ein Secret-Sharing-Verfahren heißt *robust*, wenn es den Teilnehmern möglich ist, zu kontrollieren, ob andere Teilnehmer sich an das Protokoll halten. Hierzu muß man zusätzliche Informationen in die Teilgeheimnisse einbringen. Man spricht auch von *verifizierbarem* Secret-Sharing (insbesondere, wenn auch das Verhalten des Gebers geprüft werden kann).

- Ein Secret-Sharing-Verfahren heißt *ideal*, wenn die Bitlänge des größten Anteils, der geheim gehalten werden muß, die Bitlänge des Geheimnisses nicht übersteigt. Es läßt sich zeigen, daß ideale Verfahren nicht gleichzeitig robust sein können.

- Wenn das Geheimnis in Berechnungen verwendet werden soll, dann ist es unter Umständen möglich, die Berechnung in Teilberechnungen für die Inhaber der Teilgeheimnisse aufzubrechen, so daß diese die Berechnung durchgeführen können, ohne das Geheimnis dabei rekonstruieren zu müssen. Dies wird als *Function-Sharing* bezeichnet und eignet sich insbesondere für Funktionen mit gewissen Homomorphie-Eigenschaften. Der wichtigste Spezialfall von *Function-Sharing* ist das in Kapitel 3 beschriebene Key-Sharing.

- Die Sicherheit eines verteilten Geheimnisses kann erhöht werden, wenn man die Teilgeheimnisse regelmäßig erneuert. Die alten Anteile werden dabei gelöscht, so daß ein potentieller Angreifer die benötigten Teilgeheimnisse innerhalb eines gewissen Zeitfensters erbeuten muß, da diese sonst ungültig und wertlos werden. Wichtig dabei ist, daß das eigentliche Geheimnis gleich bleibt. Ein Secret-Sharing-Verfahren, das es den Teilnehmern erlaubt, die Teilgeheimnisse zu erneuern, ohne daß Geheimnis dafür rekonstruieren zu müssen, heißt *proaktiv*.

- Der Geber stellt eine große Schwachstelle von Secret-Sharing-Verfahren dar. In Situationen, wo das Geheimnis nicht a priori feststeht, sondern auch erst während des Verfahrens erzeugt werden kann (dies ist zum Beispiel bei kryptographischen Schlüsseln der Fall), ist es möglich, auf den Geber zu verzichten. Die Teilnehmer einigen sich dann auf ein Geheimnis

(ohne es selbst zu kennen) und erhalten dabei ihre Anteile. Durch den Verlust des Gebers als vertrauenswürdige Instanz ergibt sich die Notwendigkeit eines robusten Verfahrens, um das Verhalten der anderen (nicht unbedingt vertrauenswürdigen) Teilnehmer kontrollieren zu können.

Kapitel 3

Key-Sharing

Die Sicherheit eines Kryptosystems darf nicht davon abhängen, das Verfahren geheimzuhalten. Die Sicherheit hängt nur davon ab, den Schlüssel geheimzuhalten.

August Kerckhoffs von Nieuwenhof
in *La Cryptographie militaire*

3.1 Public-Key-Kryptographie

In herkömmlichen kryptographischen Verfahren wird eine Nachricht von einem Chiffrieralgorithmus in einen Schlüsseltext umgewandelt, der nur von einem passenden Dechiffrierverfahren wieder entschlüsselt werden kann. Der Sender parametrisiert dabei die Chiffre mit einem Schlüssel, den der Empfänger ebenfalls in die Entschlüsselungsfunktion einsetzen muß. Einem Angreifer ohne Kenntnis des Schlüssels bleibt nur die Möglichkeit, alle denkbaren Schlüssel durchzuprobieren, was angesichts deren Anzahl unpraktikabel ist.

Ein Problem bei diesen Verfahren besteht in der Geheimhaltung der Schlüssel. Dies ist noch zu bewältigen, wenn die Daten nur vom Sender selbst entschlüsselt werden sollen (kryptographische Datenspeicher). Ansonsten muß der Sender allen Empfängern vorab und auf sicherem Wege den Schlüssel zukommen lassen. Wenn der Schlüssel mehrmals verwendet werden soll, müssen alle Beteiligten den Schlüssel fortan geheim halten, und den übrigen Teilnehmern zutrauen, daß diese den Schlüssel ebenfalls geheim halten.

Ein zweites Problem ist die Anzahl der benötigten Schlüssel. Wenn jeder Teilnehmer in der Lage sein soll, jedem anderen Teilnehmer eine Nachricht zu senden, deren Inhalt kein dritter Teilnehmer (und natürlich auch kein Außenstehender) erkennen kann, dann braucht jedes Paar von Teilnehmern einen eigenen Schlüssel. Die Zahl der Schlüssel wächst hierbei quadratisch mit der Anzahl der Teilnehmer. In großen Gemeinschaften (oder gar dem Internet als Ganzem) ist dies völlig unpraktikabel.

Das Problem der gemeinsam geheimzuhaltenden Schlüssel wird bei der sogenannten Public-Key- (oder asymmetrischen) Kryptographie umgangen. In diesem von Whitfield Diffie und Martin Hellman 1976 vorgestellten Verfahren [DH76] besitzt jeder Teilnehmer einen privaten und einen öffentlichen Schlüssel.

Diese beiden Schlüssel bilden dadurch, daß sie in einer gewissen mathematischen Beziehung zueinander stehen, ein Schlüsselpaar. Mit dem öffentlichen Schlüssel eines Empfängers können Nachrichten verschlüsselt werden, die anschließend nur mit dessen privatem Schlüssel wieder entziffert werden können.

Ein Public-Key-Kryptoverfahren basiert auf einem schwierigen (das heißt für praktische Zwecke nicht lösbaren) mathematischen Problem, das es unmöglich macht, den privaten Schlüssel aus dem öffentlichen Schlüssel zu berechnen, obwohl die Beziehung der beiden zueinander allgemein bekannt ist (sonst wäre es nicht als Schlüsselpaar zu verwenden). So wird es möglich, den öffentlichen Schlüssel völlig frei zu verteilen, ohne daß dadurch eine Gefahr für den geheimen Schlüssel (der weiterhin verborgen bleiben muß) entsteht.

Man kann sich die Verwendung asymmetrischer Kryptographie wie einen Hausbriefkasten vorstellen [CKLW00, S. 8]: Jedermann, der im Besitz des öffentlichen Schlüssels ist, kann einen Brief hineinwerfen. Aber nur der Empfänger als Besitzer des privaten Schlüssels kann den Briefkasten öffnen und den Brief wieder herausnehmen.

Neben dem vereinfachten Schlüsselmanagement bieten asymmetrische kryptographische Verfahren auch die Möglichkeit digitaler Signaturen. Hierbei erzeugt der Absender für das zu signierende Dokument mithilfe seines privaten Schlüssels eine Signatur, die fortan von jedem unter Verwendung des öffentlichen Absenderschlüssels überprüft werden kann.

Die bekannteste konkrete Umsetzung von Diffies und Hellmans Idee ist das nach seinen Erfindern Ronald Rivest, Fiat Shamir und Leonard Adleman benannte RSA-Verfahren [RSA78], das wir in Abschnitt 3.3 erläutern wollen. Fast überall wo heute asymmetrische Kryptographie zum Einsatz kommt handelt es sich um RSA.

3.2 Modellierung von Key-Sharing

Das wichtigste Einsatzgebiet von Secret-Sharing ist die sichere Verwahrung und Verwendung von privaten Schlüsseln eines Public-Key-Kryptosystems. Durch simple Secret-Sharing-Verfahren kann ein solcher Schlüssel sicher auf mehrere Teilhaber aufgeteilt und von diesen verwahrt werden. Soll der Schlüssel allerdings eingesetzt werden (zum Beispiel um eine digitale Signatur zu leisten), so muß er an einem Ort rekonstruiert werden, wodurch er wieder sehr angreifbar wird. Die Idee des Key-Sharings ist es, die Teilhaber in die Lage zu versetzen, die kryptographischen Operationen mit ihren Teilschlüsseln auszuführen, so daß die Ergebnisse anschließend kombiniert werden können. Dabei behalten sie die Kontrolle über ihre Teilschlüssel, die sie nicht offenbaren müssen.

Das Shamir-Verfahren und homomorphe Funktionen Eine kryptographische Operation (etwa eine Entschlüsselung oder Signatur) ist eine Funktion $g(x, k)$, deren Ergebnis von einer Eingabe x (beispielsweise der Nachricht) und einem Schlüssel k abhängt. Viele Kryptosysteme haben die Eigenschaft, daß diese Funktion homomorph ist, also

$$g(x, k_1 + k_2) = g(x, k_1) * g(x, k_2)$$

(dies muß nicht unbedingt für die gesamte Operation gelten, es reicht, wenn eine wesentliche Komponente davon homomorph ist: Hashfunktionen und Padding kann man hierbei oft außer acht lassen). Diese Homomorphie läßt sich gut mit dem Shamir-Secret-Sharing verknüpfen, denn dort wird der Schlüssel k verteilt als

$$k = \sum_{i \in \Lambda} \lambda_{i,\Lambda} k_i,$$

so daß $g(x, k)$ verteilt ausgewertet werden kann:

$$g(x,k) = g(x, \sum_{i \in \Lambda} \lambda_{i,\Lambda} k_i) = \prod_{i \in \Lambda} g(x, \lambda_{i,\Lambda} k_i) = \prod_{i \in \Lambda} g(x, k_i)^{\lambda_{i,\Lambda}}.$$

Dadurch wird es möglich, daß die Teilnehmer jeder für sich $g(x, k_i)$ berechnen und das Ergebnis an einen Kombinierer übermitteln, der das Endergebnis $g(x, k)$ bestimmen kann, wenn er genügend Teilergebnisse erhält. Dazu benötigt der Kombinierer keinerlei geheime Informationen, lediglich die Identität der Teilnehmer (ihre Nummern i).

Wir weisen darauf hin, daß wir in dieser Darstellung unterschlagen haben, daß die Berechnungen des Shamir-Verfahrens in einer primen Restklasse zu erfolgen haben, was zu einigen technischen Problem führt, auf die wir später eingehen werden.

Die bekanntesten Public-Key-Verfahren setzen auf dem RSA- (siehe Abschnitt 3.3) oder dem ElGamal-Verfahren (siehe Abschnitt 3.7) auf. Beide Verfahren erfüllen die geschilderte Homomorphie-Eigenschaft und eignen sich somit für Key-Sharing.

Ein Key-Sharing-Verfahren auf Basis eines $\binom{t}{n}$-Secret-Sharing-Verfahrens wird auch als *Threshold Cryptography* bezeichnet.

Ablauf eines Key-Sharing In der Literatur lassen sich verschiedene Modellierungen der Vorgänge und Teilnehmer eines Key-Sharing finden. In dieser Arbeit gehen wir von einer Situation aus, in der

- es einen Geber gibt, dem alle Teilnehmer vertrauen und dessen Aufgabe es ist, einen zuvor von ihm selbst oder außerhalb des Modells erzeugten privaten Schlüssel zu verteilen,
- zur Erzeugung der Teilschlüssel ein $\binom{t}{n}$-Verfahren eingesetzt wird,
- der Geber mit niemandem während der Berechnung der Teilschlüssel kommuniziert,
- der Geber die Teilschlüssel auf einem abhörsicheren Kanal an die Teilnehmer übermittelt,
- der Geber sich nach der erfolgten Übermittlung der Teilschlüssel zurückzieht,
- die Teilnehmer ihre Teilschlüssel so verwalten, als handelte es sich dabei um ihre privaten Schlüssel,

- die Teilnehmer mit ihren Teilschlüsseln anschließend gemeinsam Private-Key-Operationen (Signaturen, Entschlüsselungen) durchführen können,
- ein Klient, der eine Entschlüsselung oder Signatur wünscht, sich an die Teilnehmer wendet, die wiederum seine Identität und Berechtigung prüfen,
- die Teilnehmer bei einer solchen Anfrage eine Teilsignatur beziehungsweise -entschlüsselung berechnen, ohne dabei untereinander oder mit jemand anderem zu kommunizieren,
- die Teilnehmer ihr Ergebnis an den Klienten übermitteln, wobei bei Entschlüsselungen ein abhörsicherer Kanal zu wählen ist,
- der Klient die Ergebnisse der Teilnehmer aufgrund lediglich öffentlicher Informationen und seiner Eingabe zu einer gültigen Signatur oder Entschlüsselung kombinieren kann,
- eine gemeinsam erzeugte Signatur sich nicht von einer auf normale (nicht verteilte) Art entstandener Signatur unterscheiden läßt.

Diese Modellierung zeichnet sich im wesentlichen durch zwei Merkmale aus, nämlich die Existenz eines vertrauenswürdigen Gebers und die fehlende Interaktion zwischen den Teilnehmern. Dies hat zur Folge, daß nur wenige Nachrichten während des Key-Sharing ausgetauscht werden müssen und die Kommunikation asynchron, zum Beispiel per Email, erfolgen kann. Eine kurze Diskussion von Modellen ohne Geber findet sich in Abschnitt 3.9. Eine weitere Folge der nicht vorhandenen Interaktion ist die Unmöglichkeit der Erzeugung von verteilten Signaturen für Verfahren, die auf diskreten Logarithmen basieren, wie etwa ElGamal (Abschnitt 3.7). Dies liegt daran, daß diese Signaturen randomisiert werden müssen, so daß eine Verteilung die Synchronisation der verwendeten Zufallszahlen notwendig macht. Verteilte Entschlüsselungen bleiben weiterhin möglich.

3.3 Das RSA-Verfahren

Das erste und weitaus erfolgreichste Public-Key-Verfahren ist das von Ronald Rivest, Fiat Shamir und Leonard Adleman 1977 vorschlagene RSA-Verfahren [RSA78]. Es beruht auf dem RSA-Problem, einem mathematischen Problem, das mit dem Problem der Zerlegung großer Zahlen in ihre Primfaktoren verwandt ist. RSA hat eine enorme Verbreitung erfahren und wird heute in fast allen Public-Key-Produkten eingesetzt.

Die RSA-Annahme Die Sicherheit des RSA-Verfahrens beruht auf der Annahme, daß es keinen effizienten Algorithmus gibt, um das sogenannte RSA-Problem zu lösen.

Das RSA-Problem besteht darin, für eine gegebene ganze Zahl c und einen Moduln N eine ganze Zahl m zu bestimmen, so daß $c \equiv m^e \pmod{N}$ für einen ebenfalls bekannten Exponenten e. Dabei ist $N = pq$ das Produkt zweier großer

Primzahlen, die aber nicht bekannt sind, und e ist ein sogenannter invertierbarer Exponent modulo N (das Inverse von e ist allerdings auch nicht bekannt).

Eine Möglichkeit, die Zahl m zu bestimmen, besteht darin N zu faktorisieren. Dann läßt sich ein Element d berechnen mit $de \equiv 1 \pmod{(p-1)(q-1)}$ und es gilt $c^d = m^{ed} = m \pmod{N}$. Somit ist das RSA-Problem höchstens so schwierig wie das Faktorisierungsproblem. Da sich Mathematiker in den letzten 3000 Jahren vergeblich bemüht haben, einen effizienten Algorithmus zum Faktorisieren zu finden, kann man vermuten, daß dies wirklich schwierig ist. Das Hauptproblem der RSA-Annahme hingegen ist es, daß es eventuell eine andere Möglichkeit gibt, das RSA-Problem zu lösen als N zu faktorisieren. Selbst wenn das Faktorisierungsproblem schwer bleibt, könnte RSA somit gebrochen werden. Zwar ist kein solcher Algorithmus bisher bekannt, aber es konnte (im Gegensatz zum verwandten Rabin-Problem) auch nicht bewiesen werden, daß es keinen geben kann.

Schlüsselerzeugung Um ein RSA-Schlüsselpaar zu erzeugen, bestimmt der Schlüsselgenerator zunächst zwei zufällige Primzahlen p und q. Daraus ergibt sich der öffentliche Modul $N = pq$. Die Größe (Länge der Bitdarstellung) von N bestimmt die Sicherheit des Verfahrens (und den Berechnungsaufwand). Zur Zeit gilt 1024-bit-RSA als hinreichend sicher für die meisten Anwendungen, 2048-bit-RSA wird für extrem kritische Schlüssel gefordert. Demnach haben p und q jeweils mindestens 512 Bit (sie sollten gleich groß sein).

Als nächstes wählt der Generator einen öffentlichen Exponenten e mit

$$\mathrm{ggT}(e, (p-1)(q-1)) = 1.$$

Außer dieser Bedingung muß e keine weiteren Eigenschaften besitzen. Aus Sicherheitsgründen (Low-Exponent-Attacke) sollte e nicht zu klein sein, aber ansonsten kann es zur Beschleunigung von Public-Key-Operationen stets einer kleinen Konstante gleichgesetzt werden, etwa 65537.

Aufgrund der Tatsache, daß e und $(p-1)(q-1)$ keine gemeinsamen Teiler besitzen, kann mit dem erweiterten Euklidschen Algorithmus der geheime Exponent d bestimmt werden, mit

$$ed = 1 \pmod{(p-1)(q-1)}.$$

Der öffentliche Schlüssel besteht aus (e, N), der private Schlüssel ist (d, N) oder auch (sinnvoll zur Beschleunigung von Berechnungen von Private-Key-Operationen mit dem Chinesischen Restsatz) (d, p, q).

Schlüsselverwendung RSA-Schlüssel lassen sich für zwei Operationen einsetzen, nämlich zur Nachrichtenverschlüsselung und für digitale Signaturen. In beiden Fällen kommt dabei eine identische Berechnungsvorschrift zum Einsatz, lediglich die Rollen von öffentlichem und privatem Schlüssel werden vertauscht. Diese Symmetrie wird sich später beim RSA-Key-Sharing als nützlich erweisen.

Um eine Nachricht m für den Empfänger mit dem öffentlichen Schlüssel (e, N) zu verschlüsseln, wird sie als Zahl zwischen 1 und $N-1$ aufgefaßt und

mit dem öffentlichen Exponenten e potenziert. Es ergibt sich der Schlüsseltext c, den der Empfänger mit seinem geheimen Exponenten d entschlüsseln kann:

$$c = m^e \pmod N \qquad \text{und} \qquad m = c^d = (m^e)^d = m^{ed} = m^1 = m \pmod N.$$

Durch die Interpretation der Nachricht m als natürliche Zahl kleiner als N ergibt sich, daß die Bitlänge der Nachricht nicht länger als die des RSA-Moduln sein kann. In der Tat verschlüsselt man längere Nachrichten dann auch mit einem symmetrischen Verfahren und verschlüsselt nur den symmetrischen Schlüssel (der je nach Verfahren zwischen 50 und 200 Bits lang ist) mit RSA. Außerdem bietet es sich aus Sicherheitsgründen an, alle Nachrichten mit einigen Zufallsbits auf eine feste Länge zu bringen (Padding), so daß unter anderem jede Nachricht zu immer neuen Schlüsseltexten führt.

Neben Verschlüsselungen sind auch Signaturen möglich. Um eine Signatur für eine Nachricht m zu erzeugen, die wieder als natürliche Zahl zwischen 1 und $N-1$ angesehen wird, potenziert der Absender die Nachricht m mit seinem privaten Exponenten d. Es ergibt sich eine Signatur s, von der mit dem öffentlichen Schlüssel (e, N) überprüft werden kann, ob sie zu der Nachricht gehörte:

$$s = m^d \pmod N \qquad \text{und} \qquad m = s^e = (m^d)^e = m^{de} = m^1 = m \pmod N.$$

Auch hier ist wieder anzumerken, daß Nachrichten üblicherweise größer sind als die RSA-Moduln, so daß anstelle der Nachricht m nur eine Prüfsumme $h(m)$ signiert wird, die mithilfe einer kryptographischen Hashfunktion gewonnen werden kann. Dadurch wird es allerdings unmöglich, aus der Signatur die Nachricht zurückzugewinnen. Vielmehr muß die Nachricht zusammen mit der Signatur übertragen werden. Ein Padding mit Zufallszahlen ist bei RSA-Signaturen nicht angezeigt, so daß die Signatur ein deterministisches Verfahren bleibt (dies ermöglicht die in den folgenden Abschnitten beschriebenen verteilten RSA-Signaturen).

3.4 Einfaches RSA-Key-Sharing

Das RSA-Verfahren hat die erstaunliche Eigenschaft, daß es auf einem Problem beruht, daß sehr einfach zu erklären, aber offenbar sehr schwer zu lösen ist. Der größte Vorteil, der sich daraus ergibt, sind die sehr soliden Argumente für die Sicherheit von RSA, die wesentlich besser fundiert sind als diejenigen für weniger transparente Verfahren, wie etwa die symmetrische DES-Chiffre. Ein anderer Vorteil ist die Möglichkeit, andere Kryptosysteme auf RSA aufzubauen, zum Beispiel das in diesem Abschnitt vorgestellte RSA-Key-Sharing.

Schlüsselerzeugung Um einen geheimen RSA-Schlüssel (d, p, q) auf n Teilnehmer aufzuteilen, die ihn dann nur gemeinsam verwenden können, wählt der Geber für jeden Teilnehmer i zufällige Exponenten d_i aus der Menge $\{1 \ldots (p-1)(q-1)\}$, deren Summe (modulo $(p-1)(q-1)$) wieder den eigentlichen Exponenten d ergibt:

$$\sum_i d_i = d \pmod{(p-1)(q-1)}.$$

Den Teilnehmern übermittelt er die Teilschlüssel (d_i, N) (es ist absolut notwendig, den Teilnehmern die Faktorisierung $N = pq$ zu verheimlichen, da sie ansonsten den geheimen Exponenten d berechnen könnten).

Schlüsselverwendung Da RSA-Berechnungen einfache Exponentiationen sind, gelten die allgemein bekannten Rechenregeln hierzu, insbesondere auch die Homomorphie

$$m^{d_1+d_2} = m^{d_1} m^{d_2}.$$

Damit ist auch klar, daß die Teilschlüssel (d_i, N) dazu verwendet werden können, um Teilergebnisse s_i der Berechnung $s = m^d \pmod N$ zu erzeugen, die dann durch Multiplikationen zum Gesamtergebnis kombiniert werden können:

$$s_i = m^{d_i} \pmod N \qquad \text{und} \qquad s = \prod_i s_i = \prod_i m^{d_i} = m^{\sum_i d_i} = m^d \pmod N.$$

Wir bemerken hierzu, daß die Teilberechnungen sich (außer durch den Wert des Exponenten) nicht von einer normalen RSA-Berechnung unterscheiden und daß zur Kombination der Teilergebnisse (durch Multiplikation modulo N) keine geheimen Informationen notwendig sind.

3.5 Redundantes RSA-Key-Sharing

Wie wir im vorigen Abschnitt gesehen haben, erlauben die mathematischen Eigenschaften eines RSA-Schlüssels, ihn auf recht einfache Weise in eine beliebige Anzahl von Schlüsselanteilen zu zerlegen, die unabhängig voneinander zum Erzeugen von Teilergebnissen verwendet werden können, aus denen dann das Ergebnis der privaten RSA-Operation abgeleitet werden kann, ohne daß der eigentliche Schlüssel jemals rekonstruiert werden muß. Da das geschilderte Verfahren jedoch stets alle Teilschlüssel benötigt, bricht es bereits beim Verlust eines einzigen dieser Teile zusammen. In diesem Abschnitt wollen wir zeigen, wie sich das Shamir-Verfahren aus Abschnitt 2.2.1 auf RSA-Teilschlüssel übertragen läßt und so ein Threshold-RSA-Key-Sharing ermöglicht.

3.5.1 Verfahren von Frankel, Gemmell, MacKenzie und Yung

Die direkteste Adaption des Shamir-Verfahrens für den Einsatz mit RSA-Exponenten besteht darin, daß man für ein $\binom{t}{n}$-Key-Sharing versucht, den geheimen Exponenten d durch ein Polynom

$$f(x) = d + a_1 x + \ldots + a_{t-1} x^{t-1} \pmod{(p-1)(q-1)}$$

und dessen Interpolationsstellen

$$d_i = f(i) \pmod{(p-1)(q-1)}$$

auf mehrere RSA-Teilschlüssel (d_i, N) zu verteilen. Dann könnte man genau wie im vorigen Abschnitt RSA-Teilberechnungen mit den Teilschlüsseln ausführen und anschließend durch Multiplikation zum Gesamtergebnis kombinieren: um

gemeinsam $s = m^d \pmod N$ zu berechnen, bestimmt jeder Teilnehmer $i \in \Lambda$ seinen Anteil

$$s_i = m^{d_i \lambda_{i,\Lambda}} \pmod N$$

mit

$$\lambda_{i,\Lambda} = \prod_{l \in \Lambda \setminus \{i\}} \frac{l}{l-i} \pmod{(p-1)(q-1)}$$

und es ergibt sich

$$s = \prod_{i \in \Lambda} s_i = m^d \pmod N.$$

Diese Vorgehensweise scheitert allerdings an der Berechnung der Interpolationskoeffizienten $\lambda_{i,\Lambda}$, da hierzu Invertierungen von $l-i$ modulo $(p-1)(q-1)$ notwendig sind, die Primzahlen p und q jedoch niemandem bekannt sein dürfen, da darauf die Sicherheit des RSA-Schlüssels beruht (des weiteren sind auch nicht alle $l-i$ invertierbar, beispielsweise sind es sämtliche geraden Differenzen nicht).

Schlüsselerzeugung Das Problem läßt sich jedoch durch geeignete Modifikationen des Polynoms beheben [FGPY97, MSY00]. Durch die Hinzunahme von hinreichend großen Faktoren können die Berechnungen der Koeffizienten durch normale Ganzzahldivisionen geleistet werden, so daß Invertierungen in der Exponentengruppe unbekannter Ordnung nicht notwendig sind.

Der Faktor, durch den anschließend alle Teilexponenten ohne Rest dividierbar sein müssen, hängt von der Anzahl n der Teilnehmer ab: Sei

$$L = (n-1)!.$$

Wenn die d_i Vielfache von L sind, dann können die Berechnungen

$$d_i \lambda_{i,\Lambda} = d_i \prod_{l \in \Lambda \setminus \{i\}} \frac{l}{l-i}$$

über den ganzen Zahlen erfolgen (in der ursprünglichen Arbeit [FGPY97] wurde der Faktor $L = n!$ gewählt, es genügt aber auch das kleinere $L = (n-1)!$ [MSY00]). Um dies zu erreichen, müssen alle Koeffizienten des Polynoms Vielfache von L sein, inklusive des geheim zu haltenden Achsenabschnittes a_0, der folglich nicht direkt dem Exponenten d entsprechen kann (da dieser nicht unbedingt ein Vielfaches von L sein muß). Daher bestimmen wir ein modifiziertes Geheimis a_0, aus dem sich der Exponent d zurückgewinnen läßt. Sei zunächst

$$H = \mathrm{ggT}(e, L^2).$$

Weil e invertierbar ist modulo $(p-1)(q-1)$, ist auch H invertierbar. Außerdem ist H ein Teiler von L^2 und es existieren keine gemeinsamen Teiler von e und $\frac{L^2}{H}$. Mithilfe des erweiterten Euklidschen Algorithmus können wir Faktoren P und s berechnen, für die gilt

$$eP + \frac{L^2}{H} s = 1.$$

Auf diese Weise ergibt sich eine Zahl k mit

$$d = P + L^2 k \pmod{(p-1)(q-1)} \qquad \text{und} \qquad k = dsH^{-1} \pmod{(p-1)(q-1)},$$

mit deren Hilfe wir nun einen modifizierten Exponenten

$$d' = L^2 k$$

definieren, welcher durch L teilbar ist (weil wir nicht modulo $(p-1)(q-1)$ reduzieren) und aus dem sich der ursprüngliche Exponent d zurückrechnen läßt (zumindest innerhalb der Exponentenrestklasse):

$$d = P + d' \pmod{(p-1)(q-1)}.$$

Dieser modifizierte RSA-Exponent d' wird nun durch das Polynom

$$f(x) = d' + a_1 x + \ldots + a_{t-1} x^{t-1} \qquad \text{mit} \quad a_i \in_R \{0, L, \ldots, 2L^3 n^{2+\epsilon} t\}$$

und Interpolationsstellen $d_i = f(i)$ auf die Teilnehmer verteilt. Wir erkennen, daß dieses Polynom über den ganzen Zahlen (und nicht in einer Restklasse) ausgewertet wird, und alle Interpolationsstellen d_i Vielfache von L sind.

Schlüsselverwendung Um gemeinsam $s = m^d \pmod{N}$ zu berechnen, bestimmt jeder Teilnehmer $i \in \Lambda$ seinen Anteil

$$s_i = m^{d_i \lambda_{i,\Lambda}} \pmod{N}$$

mit

$$\lambda_{i,\Lambda} = \prod_{l \in \Lambda \setminus \{i\}} \frac{l}{l-i}.$$

Aufgrund der Konstruktion der d_i sind hierfür keine Invertierungen notwendig. Es ergibt sich

$$s = m^P \prod_{i \in \Lambda} s_i = m^{P + \sum_i d_i \lambda_{i,\Lambda}} = m^{P+d'} = m^d \pmod{N}.$$

Für den letzten Schritt, in dem die partiellen Berechnungsergebnisse kombinert werden, ist eine weitere Exponentiation m^P notwendig. Hierzu schlagen die Autoren des Verfahrens vor, daß einer der Teilnehmer m^P berechnet und (zusammen mit seiner Teilberechnung m^{d_i}) an den Kombinierer übermittelt [MSY00]. Eine Alternative hierzu besteht darin, daß einer der Teilnehmer (zum Beispiel derjenige mit der kleinsten Teilnehmernummer) anstelle von m^{d_i} direkt m^{P+d_i} ermittelt und weitergibt. Dann kann der Kombinierer wie im Verfahren aus Abschnitt 3.4 einfach die Anteile zusammenmultiplizieren. In jedem Fall sind weder m noch P geheime Informationen und folglich könnte der Kombinierer den Korrekturfaktor m^P auch selbst berechnen.

3.5.2 Eine Variante des Verfahrens

Das im vorigen Abschnitt geschilderte Verfahren hat einige Nachteile, die wir hier diskutieren wollen. Abschließend stellen wir eine eigene Variante vor, die zumindest zwei dieser Nachteile behebt.

- es existiert kein Sicherheitsbeweis für das vorgestellte Verfahren. In der Tat wird es von Frankel, Gemmell, MacKenzie und Yung lediglich als Heuristik bezeichnet [FGPY97]. Darauf aufbauend entwickeln sie noch ein weiteres Verfahren, dessen Sicherheit auf das RSA-Verfahren zurückgeführt werden kann, welches allerdings Interaktionen zwischen den Teilnehmern benötigt, weshalb wir uns in dieser Arbeit nicht weiter damit beschäftigen wollen. Trotz der fehlenden beweisbaren Sicherheit ist das Verfahren von anderen Autoren aufgegriffen worden [MSY00] und wurde zum Beispiel auch im Rahmen eines von der TU Darmstadt und der japanischen Telefongesellschaft NTT gemeinsam entwickelten verteilten RSA-Zeitstempeldienstes eingesetzt [ABF$^+$99].

- die Schlüsselanteile, die den Teilnehmern zugestellt werden, sind keine normalen RSA-Schlüssel. Die Exponenten, die bei den partiellen RSA-Berechnungen zum Einsatz kommen, sind aufgrund der Multiplikationen zum Herstellen der Teilbarkeit um einige Bit länger als normale RSA-Exponenten und aufgrund der Interpolationskoeffizienten teilweise auch negativ. Beides erschwert den Einsatz von kryptographischen Bibliotheken oder Hardwareimplementierungen für RSA-Berechnungen. Insbesondere entfällt die Möglichkeit der sicheren Speicherung der Teilschlüssel auf RSA-fähigen Chipkarten.

- zum Berechnen der Teilergebnisse müssen die Teilnehmer der Berechnung bekannt sein (die Menge Λ). Wenn sich erst später herausstellt, daß einige Teilnehmer nicht zur Verfügung stehen, müssen alle Teilnehmer die Berechnung wiederholen. Das Problem vergrößert sich noch in Anbetracht des nächsten Kritikpunktes.

- ein vorliegendes Teilergebnis läßt sich nicht auf Korrektheit überprüfen. Ein fehlerhaftes Ergebnis wird erst nach der Kombination aller Teilergebnisse erkannt, ohne daß allerdings bekannt wäre, welche der Teilergebnisse zum Fehler geführt haben. Dies macht es sehr schwierig, einen fehlerhaft arbeitenden Teilnehmer zu erkennen und auszuschließen.

In unserer Implementierung haben wir eine Variation des Verfahrens entwickelt, bei der den Teilnehmern gewöhnliche RSA-Schlüssel zugeteilt werden, mit denen sie die partiellen Berechnungen durchführen können. Die Teilergebnisse können dann in beliebigen Kombinationen zusammengeführt werden, ohne daß Neuberechnungen durch die Teilnehmer notwendig werden. Damit sind zwei der angesprochenen vier Mängel behoben. Das Problem einen fehlerhaften Beitrag erkennen zu können wird ebenfalls gemildert, da alle zulässigen Kombinationen getestet werden können (ohne weitere Beteiligung der eventuell betrügerischen Teilnehmer). Bestehen bleibt die unbewiesene Sicherheit des

Verfahrens (ein beweisbar sicheres Verfahren, das auch betrügerische Teilnehmer aufdeckt, wird in Abschnitt 3.6 vorgestellt).

Schlüsselerzeugung Die Schlüsselerzeugung erfolgt zunächst genauso wie beim Frankel-Gemmell-MacKenzie-Yung-Verfahren aus Abschnitt 3.5.1. Dieses wird vollständig ausgeführt, so daß im Ergebnis die Interpolationsstellen d_i für die einzelnen Teilnehmer berechnet wurden. Wie oben dargelegt sind alle d_i durch L teilbar um Invertierungen zu vermeiden. Gleichzeitig sind die d_i dadurch auch größer als normale RSA-Exponenten und können durch die Interpolationskoeffizienten $\lambda_{i,\Lambda} = \prod_{l \in \Lambda \setminus \{i\}} \frac{l}{l-i}$, mit denen sie multipliziert werden, auch negativ werden.

In unserem Verfahren führen wir alle Divisionen, die sich durch die Koeffizienten ergeben können, bereits als Vorberechnung aus:

$$d'_i = \frac{d_i}{\prod_{l \in \{1 \ldots n\} \setminus \{i\}} l - i}$$

Infolge dessen müssen durch den einzelnen Teilnehmer i zur Berechnung seines Teilergebnisses $s_i = m^{d_i \lambda_{i,\Lambda}} \pmod N$ keine Divisionen, sondern nur noch Multiplikationen des neuen Exponenten d'_i vorgenommen werden:

$$d_i \lambda_{i,\Lambda} = d'_i \prod_{l \in \Lambda \setminus \{i\}} l \prod_{l \in \{1 \ldots n\} \setminus \Lambda} l - i$$

Diese Multiplikationen im Exponenten können aber auch als Exponentiationen ausgeführt werden:

$$m^{d_i \lambda_{i,\Lambda}} = (m^{d'_i})^{\prod_{l \in \Lambda \setminus \{i\}} l \prod_{l \in \{1 \ldots n\} \setminus \Lambda} l - i} \pmod N.$$

Schlüsselverwendung Da für die Exponentiationen keine geheimen Informationen notwendig sind, sondern lediglich Kenntnis über die Beteiligten $i \in \Lambda$, können diese Berechnungen auch beim Kombinieren durchgeführt werden. Es wird also möglich, daß die Teilergebnisse von den Teilnehmern ohne Kenntnis der Identität der anderen Beteiligten erstellt werden. Sie müssen nur noch eine gewöhnliche RSA-Operation

$$s'_i = m^{d'_i} \pmod N$$

durchführen. Dadurch können wir bei der Schlüsselerzeugung die Exponenten d'_i, die im allgemeinen zwar kleiner als die ursprünglichen d_i, aber immer noch größer als normale RSA-Exponenten, sowie teilweise negativ sein werden, modulo $(p-1)(q-1)$ reduzieren (die Gruppenordnung ist zum Zeitpunkt der Schlüsselerzeugung noch bekannt). Auf diese Weise können den Teilnehmern gewöhnliche RSA-Schlüssel übergeben werden. Die Kombination der Teilergebnisse erfolgt dann als

$$s_i = (s'_i)^{\prod_{l \in \Lambda \setminus \{i\}} l \prod_{l \in \{1 \ldots n\} \setminus \Lambda} l - i} \pmod N \qquad \forall i \in \Lambda$$

und

$$s = m^P \prod_{i \in \Lambda} s_i \pmod{N}.$$

3.6 Verifizierbares RSA-Key-Sharing

Das in Abschnitt 3.5 vorgestellte Verfahren verfügt über zwei Nachteile, nämlich den fehlenden Sicherheitsbeweis und die fehlende Möglichkeit, die von den Teilnehmern abgegebenen Berechnungsergebnisse zu verifizieren. Beide Mängel sind in einem von Victor Shoup vorgeschlagenen Verfahren nicht anzutreffen (für den Sicherheitsbeweis verweisen wir auf die Originalarbeit [Sho00]). Es erzeugt ebenfalls normale RSA-Signaturen und -Entschlüsselungen, ohne Interaktionen zwischen den Teilnehmern zu benötigen. Allerdings setzt das Verfahren spezielle RSA-Moduln voraus und die Teilschlüssel und die damit durchzuführenden Berechnungen lassen sich (im Gegensatz zu unserem Vorschlag aus Abschnitt 3.5.2) nicht in normale RSA-Implementierungen einbetten.

Schlüsselerzeugung Der Geber bestimmt zwei zufällige Primzahlen p und q, die der zusätzlichen Eigenschaft genügen, daß $p = 2p' + 1$ und $q = 2q' + 1$ für zwei weitere Primzahlen p' und q' (Man nennt solche Primzahlen p und q starke Primzahlen). Dadurch ergibt sich der RSA-Modul $N = pq$. Den öffentlichen Exponenten e wählt der Geber als beliebige Primzahl $e > n$. Für den Sicherheitsbeweis ist es wichtig, daß alle Teilberechnungen nicht über der ganzen Gruppe $(\mathbb{Z}/N\mathbb{Z})$ausgeführt werden, sondern nur über den Quadraten aus $(\mathbb{Z}/N\mathbb{Z})^*$. Die Quadrate aus $(\mathbb{Z}/N\mathbb{Z})^*$bilden eine zyklische Untergruppe $\mathbb{Q}_N$der Ordnung $p'q'$, so daß die Exponenten für Elemente aus $\mathbb{Q}_N$modulo $p'q'$ gerechnet werden können. Folglich wird der private Exponent d auch derart bestimmt, daß $de = 1 \pmod{p'q'}$ gilt (und nicht modulo $(p-1)(q-1)$). Für diesen Exponenten d wird nun wie beim Shamir-Verfahren ein Polynom

$$f(x) = d + a_1 x + \ldots + a_{t-1} x^{t-1} \pmod{p'q'} \qquad \text{mit} \qquad a_i \in_R \{1 \ldots p'q' - 1\}$$

konstruiert. Wir stellen fest, daß hier nicht über einem Primkörper gerechnet wird, so daß nicht alle Elemente invertierbar sind. Es wird sich aber zeigen, daß das Verfahren keine Invertierungen benötigt.

Sei nun $L(N)$ die Bitlänge von N und L_1 die Bitlänge der Ausgabe einer kryptographischen Hashfunktion (also beispielsweise 160). Für jeden Teilnehmer wählt der Geber die Teilschlüssel d_i zufällig aus der Menge

$$d_i \in_R \{x : 0 \leq x < 2^{L(n)+L_1}, x \equiv f(i) \pmod{p'q'}\}.$$

Um die Berechnungen der Teilnehmer später verifizieren zu können, wählt der Geber weiterhin ein $v \in_R \mathbb{Q}_N$ (also ein invertierbares Quadrat aus $(\mathbb{Z}/N\mathbb{Z})$) und bestimmt für jeden Teilnehmer i dessen Verifikationsschlüssel $v_i = v^{d_i} \in \mathbb{Q}_N$.

Der öffentliche Schlüssel ist (e, N), der geheime Anteil von Teilnehmer i ist d_i. Dazu kommen die (öffentlichen) Verifikationsschlüssel v und v_i.

Schlüsselverwendung Es stellt sich wieder das Problem der Berechnung der Lagrange-Koeffizienten

$$\lambda_{i,\Lambda} = \prod_{l \in \Lambda \setminus \{i\}} \frac{l}{l-i}.$$

Ähnlich wie beim Verfahren aus Abschnitt 3.5.1 kann man diese Berechnungen über den ganzen Zahlen durchführen, wenn man den Ausdruck mit $L = (n-1)!$ zu

$$\lambda'_{i,\Lambda} = L\lambda_{i,\Lambda} = L \prod_{l \in \Lambda \setminus \{i\}} \frac{l}{l-i} \in \mathbb{Z}$$

erweitert (in der Originalarbeit [Sho00] ist $L = n!$, aber auch hier genügt wieder $L = (n-1)!$). Mit diesen modifizierten Koeffizienten kann man zwar nicht direkt den Exponenten d rekonstruieren, aber zumindest

$$Ld \equiv \sum_{i \in \Lambda} \lambda'_{i,\Lambda} d_i \pmod{p'q'}.$$

Die von jedem Teilnehmer durchgeführte Berechnung ist

$$s_i = m^{2Ld_i} \in \mathbb{Q}_N$$

und liefert ein Element aus $\mathbb{Q}_N$zurück, wobei allerdings vorausgesetzt wird, daß die Nachricht m aus $(\mathbb{Z}/N\mathbb{Z})^*$stammt. Diese zusätzliche Forderung gegenüber dem normalen RSA-Verfahren ist allerdings keine wirkliche Einschränkung, da fast alle Elemente aus $(\mathbb{Z}/N\mathbb{Z})$ebenfalls in $(\mathbb{Z}/N\mathbb{Z})^*$liegen. Wenn man ein m aus $(\mathbb{Z}/N\mathbb{Z})\setminus(\mathbb{Z}/N\mathbb{Z})^*$ findet, so gilt $\mathrm{ggT}(m, N) \neq 1$ und man hat N faktorisiert. Bei digitalen Signaturen kann das Problem ebenfalls nicht auftreten, da die Hashwerte von Nachrichten kürzer sind als die Bitlängen von p und q.

Neben den Teilberechnungen s_i erzeugen die Teilnehmer der Rekonstruktion mithilfe der Verifikationsschlüssel noch einen Korrektheitsbeweis. Bevor wir diesen Schritt erläutern, wollen wir jedoch zeigen, wie die s_i zu einem s kombiniert werden können, so daß gilt $s^e = m$ (damit hätten wir eine gültige RSA-Signatur, oder analog eine Entschlüsselung). Zur Kombination berechnen wir

$$w = \prod_{i \in \Lambda} s_i^{2\lambda'_{i,\Lambda}} = m^{4L^2 d} \pmod{N}.$$

Das Endergebnis ist dann

$$s = w^a m^b \pmod{N} \qquad \text{mit} \qquad a(4L^2) + be = 1,$$

wie die Rechnung

$$s^e = w^{ae} m^{be} = m^{4L^2 dae} m^{be} = m^{4L^2 a(de)+be} = m^{a(4L^2)+be} = m \pmod{N}$$

zeigt (die Exponenten a und b existieren, da $\mathrm{ggT}(e, L) = 1$, weil $e > n$ eine Primzahl ist).

Mit dem Korrektheitsbeweis können die Teilnehmer nachweisen, daß der diskrete Logarithmus von s_i^2 zur Basis m^{4L} der gleiche ist wie der diskrete

Logarithmus von v_i zur Basis v (also d_i). Natürlich darf dabei keine zusätzliche Information über d_i entstehen (die Beziehung $v_i = v^{d_i}$ ist ja bereits bekannt). Sei H eine Hashfunktion mit einer Ausgabe von L_1 Bits. Der Teilnehmer i wählt eine zufällige Zahl $r \in_R \{0 \dots 2^{L(N)+3L_1} - 1\}$ (da er die Gruppenordnung $p'q'$ nicht kennt, muß er mit hinreichend großen Zahlen arbeiten) und berechnet den Korrektheitsbeweis (z, c) als

$$c = H(v, m^{4L}, v_i, s_i^2, v^r, m^{4Lr}) \qquad \text{und} \qquad z = d_i c + r.$$

Damit kann die korrekte Berechnung von s_i^2 überprüft werden, denn es muß

$$c = H(v, m^{4L}, v_i, s_i^2, v^z v_i^{-c}, m^{4Lz} s_i^{-2c})$$

gelten (daß mit s_i^2 anstatt s_i gerechnet wird, wird für den Sicherheitsbeweis benötigt).

3.7 Das ElGamal-Verfahren

Neben dem RSA-Verfahren ist das auf der Schwierigkeit der Berechnung diskreter Logarithmen beruhende ElGamal-Verfahren [ElG85] die bekannteste Basis für Public-Key-Kryptosysteme. Es wurde 1985 von Taher ElGamal vorgeschlagen und ähnelt dem Diffie-Hellman-Schlüsselaustausch-Protokoll.

Das Diffie-Hellman-Problem Die Sicherheit des ElGamal-Verfahrens beruht auf der Annahme, daß es keinen effizienten Algorithmus gibt, um das sogenannte Diffie-Hellman-Problem zu lösen.

Das Diffie-Hellman-Problem besteht darin, für eine gegebene prime Restklassengruppe $(\mathbb{Z}/p\mathbb{Z})^*$mit einem bekannten erzeugenden Element (Generator) $g \in (\mathbb{Z}/p\mathbb{Z})^*$ für zwei Elemente $A = g^a$ und $B = g^b$ das Element $C = g^{ab}$ zu berechnen, obwohl a und b unbekannt sind.

Eine Möglichkeit, das Element C zu bestimmen, besteht darin die Exponenten a oder b zu berechnen. Dann ergibt sich $C = A^b = B^a$. Man bezeichnet a und b als diskrete Logarithmen von A beziehungsweise B zur Basis g in der Gruppe $(\mathbb{Z}/p\mathbb{Z})^*$. Für dieses Diskrete-Logarithmus-Problem sind allerdings keine effizienten Algorithmen bekannt. Das Diffie-Hellman-Problem und das Diskrete-Logarithmus-Problem stehen dabei in einer ähnlichen Beziehung zueinander wie das RSA-Problem und das Faktorisierungs-Problem: es ist nicht bekannt, ob sie äquivalent sind.

Da das Diffie-Hellman-Problem in keiner bekannten Beziehung zum RSA-Problem steht, bietet das ElGamal-Verfahren eine echte Alternative zum RSA-Verfahren: selbst wenn eines der Verfahren aufgrund neuer Erkenntnisse unsicher werden sollte, muß das andere davon nicht betroffen sein. Darüberhinaus läßt sich das ElGamal-Verfahren leicht auf andere Gruppen als $(\mathbb{Z}/p\mathbb{Z})^*$übertragen, in denen das Diffie-Hellman- und das Diskrete-Logarithmus-Problem andere Lösungsansätze erfordern. Am bekanntesten sind hierbei die Elliptischen Kurven, deren Struktur kürzere Schlüssellängen (im Vergleich zu ElGamal über $(\mathbb{Z}/p\mathbb{Z})^*$) zulassen.

Schlüsselerzeugung Um ein ElGamal-Schlüsselpaar zu erzeugen, wählt der Schlüsselgenerator eine Primzahl p und bestimmt eine Primitivwurzel g aus $(\mathbb{Z}/p\mathbb{Z})^*$(dies ist ein Element aus $(\mathbb{Z}/p\mathbb{Z})^*$, dessen Potenzen die ganze Gruppe $(\mathbb{Z}/p\mathbb{Z})^*$erzeugen). Dann wählt er zufällig und gleichverteilt den geheimen Exponenten a aus $\{1 \dots p-2\}$ und berechnet den öffentlichen Schlüssel (p, g, A) mit

$$A = g^a \pmod{p}.$$

Die Größe von p bestimmt die Sicherheit des Verfahrens. Es muß unmöglich gemacht werden, diskrete Logarithmen in $(\mathbb{Z}/p\mathbb{Z})$zu berechnen. Zur Zeit gelten 1024-Bit-Schlüssel als hinreichend sicher.

Schlüsselverwendung Wie beim RSA-Verfahren können auch ElGamal-Schlüssel für zwei unterschiedliche Operationen, nämlich Verschlüsselung und Signatur herangezogen werden. Im Unterschied zu RSA sind hierbei jedoch zwei verschiedene Algorithmen anzuwenden.

Um eine Nachricht m für den Empfänger mit dem öffentlichen Schlüssel (p, g, A) zu verschlüsseln, wählt der Absender eine Zufallszahl $b \in_R \{1 \dots p-2\}$ und berechnet

$$B = g^b \pmod{p}.$$

Anschließend wird die Nachricht als Zahl zwischen 1 und $p-1$ interpretiert und es ergibt sich der Schlüsseltext (B, c) mit

$$c = A^b m \pmod{p}.$$

Der Empfänger kann die Nachricht unter Verwendung seines geheimen Exponenten a entschlüsseln:

$$B^{p-1-a}c = B^{-a}c = g^{b(-a)}A^b m = g^{-ab}g^{ab}m = m \pmod{p}.$$

Genau wie bei der RSA-Verschlüsselung wird man lange Nachrichten zunächst symmetrisch verschlüsseln und nur den symmetrischen Schlüssel mit ElGamal übermitteln. Im Gegensatz zu RSA ist die ElGamal-Verschlüsselung durch die Wahl von b randomisiert, so daß ein Padding mit Zufallszahlen nicht notwendig ist (die Sicherheit des Verfahrens beruht übrigens darauf, daß bei jeder Verschlüsselung ein neues b gewählt wird).

Um eine Signatur für die Nachricht m zu erzeugen, benutzt der Absender mit dem privaten Schlüssel (p, g, a) eine kryptographische Hashfunktion, die die Nachricht auf einen Wert $h(m)$ zwischen 1 und $p-2$ abbildet. Anschließend wählt er eine Zufallszahl $k \in_R \{1 \dots p-2\}$ mit $\mathrm{ggT}(k, p-1) = 1$ und berechnet das Inverse k^{-1} von k modulo $p-1$ sowie

$$r = g^k \pmod{p} \qquad \text{und} \qquad s = k^{-1}(h(x) - ar) \pmod{p-1}.$$

Die Signatur ist das Paar (r, s). Bei der Verifikation kann mit dem öffentlichen Schlüssel (p, g, A) überprüft werden, ob gilt

$$A^r r^s = g^{h(x)} \pmod{p}.$$

In diesem Fall handelt es sich um eine gültige Signatur, denn

$$A^r r^s = g^{ar} g^{kk^{-1}(h(x)-ar)} = g^{ar-ar+h(x)} = g^{h(x)} \pmod{p}.$$

3.8 ElGamal-Key-Sharing

Wie wir gesehen haben, sind ElGamal-Verschlüsselungen und -Signaturen randomisierte Berechnungen, in die jeweils eine Zufallszahl eingeht. Die Verschlüsselung ist eine Public-Key-Operation, so daß ein Key-Sharing auf sie keinen Einfluß hat. Zur Signatur allerdings wird der private Schlüssel (beziehungsweise die privaten Schlüsselteile) benötigt, so daß die Erzeuger der Teilsignaturen untereinander kommunizieren müssen, um sich auf die Zufallszahl zu einigen. Da wir uns in dieser Arbeit nur mit Protokollen beschäftigen wollen, die keine Interaktion der Teilnehmer erforderlich machen, werden wir verteilte ElGamal-Signaturen (und andere, davon abgeleitete Signaturen, beispielsweise DSA) nicht besprechen und verweisen auf die Literatur [GJKR96].

Genau wie beim nicht-redundanten RSA-Key-Sharing ist es sehr einfach möglich, den geheimen Exponenten a in beliebig viele Summanden a_i mit $\sum_i a_i = a \pmod{p-1}$ zu zerlegen, die dann unabhängig voneinander zur Berechnung von Teilentschlüsselungen verwendet werden können:

$$B_i = B^{-a_i} \pmod{p} \qquad \text{und} \qquad c\prod_i B_i = Bc = m \pmod{p}.$$

Beim Versuch, das Shamir-Verfahren direkt auf ElGamal-Exponenten zu übertragen, stößt man wie beim RSA-Verfahren erneut auf das Problem, daß sich die Exponenten in einer nicht-primen Restgruppe bewegen, in diesem Fall $(\mathbb{Z}/(p-1)\mathbb{Z})$ (die Ordnung dieser Gruppe muß zwar im Gegensatz zum RSA-Verfahren nicht geheim gehalten werden, aber das ändert nichts daran, daß sich gewisse Elemente nicht invertieren lassen). Es gibt verschiedene Vorschläge, wie dieses Problem umgangen werden kann. Man kann das ElGamal-Verfahren in anderen Zahlkörpern durchführen, etwa $(\mathbb{Z}/p\mathbb{Z})$mit $p = 2^l$ und $p-1$ prim, ein anderes Secret-Sharing-Verfahren anwenden [DF90] oder ähnlich wie im Pedersen-Verfahren aus Abschnitt 2.3.1 in einer primen Untergruppe G_q von $(\mathbb{Z}/p\mathbb{Z})^*$rechnen [Ped91]. Wir verwenden hier letztere Vorgehensweise.

Wenn die Primzahl p sich als $p = mq + 1$ darstellen läßt, wobei q ebenfalls eine Primzahl ist, dann läßt sich das ElGamal-Verfahren wie folgt abwandeln: Das Element g wird nicht mehr so gewählt, daß es die gesamte Gruppe $(\mathbb{Z}/p\mathbb{Z})^*$erzeugt, sondern lediglich eine Untergruppe G_q mit q Elementen. Infolge dessen bewegen sich die auftretenden Exponenten nur noch in der (kleineren, aber ebenfalls primen) Gruppe $(\mathbb{Z}/q\mathbb{Z})$, so daß man im Exponenten nicht mehr modulo $p-1$ rechnen muß, sondern modulo q rechen kann (in der Basis muß weiterhin modulo p gerechnet werden). Auf diese Weise kommt man mit kleineren Exponenten aus (das wird beim DSA-Verfahren ausgenutzt) und kann auch das Shamir-Verfahren direkt verwenden.

Schlüsselerzeugung Der Schlüsselgenerator wählt zwei Primzahlen p und q mit $p = mq+1$ und bestimmt eine Element g aus $(\mathbb{Z}/p\mathbb{Z})^*$mit Ordnung q (dieses Element erzeugt eine Untergruppe G_q von $(\mathbb{Z}/p\mathbb{Z})^*$). Dann wählt er zufällig und gleichverteilt den geheimen Exponenten a aus $\{1 \dots q-1\}$ und berechnet den öffentlichen Schlüssel (p, q, g, A) mit

$$A = g^a \pmod{p}.$$

Die geheimen Teilexponenten a_i für die Teilnehmer werden gemäß dem Shamir-Verfahren durch ein Polynom $f(x)$ über $(\mathbb{Z}/q\mathbb{Z})$erzeugt:

$$f(x) = a + f_1 x + \ldots + f_{t-1} x^{t-1} \pmod{q} \qquad \text{mit} \quad f_i \in_R \{0, \ldots, q-1\}$$

und

$$a_i = f(i).$$

Schlüsselverwendung Um eine Nachricht m für den Empfänger mit dem öffentlichen Schlüssel (p, q, g, A) zu verschlüsseln, wählt der Absender eine Zufallszahl $b \in_R \{1 \ldots q-1\}$ und berechnet

$$B = g^b \pmod{p}.$$

Anschließend wird die Nachricht als Zahl zwischen 1 und $p-1$ interpretiert und es ergibt sich der Schlüsseltext (B, c) mit

$$c = A^b m \pmod{p}.$$

Um gemeinsam $m = B^{q-a} c \pmod{p}$ zu berechnen, bestimmt jeder Teilnehmer $i \in \Lambda$ seinen Anteil

$$B_i = c^{a_i} \pmod{p}$$

und die Anteile können dann kombiniert werden, um die Nachricht zu entschlüsseln:

$$m = Bc = \prod_{i \in \Lambda} B_i^{\lambda_{i,\Lambda}} c \pmod{p}$$

mit

$$\lambda_{i,\Lambda} = \prod_{l \in \Lambda \setminus \{i\}} \frac{l}{l-i} \pmod{q}.$$

Die notwendigen Invertierungen zur Berechnung von $\lambda_{i,\Lambda}$ sind nicht schwierig, da der Modul q sowohl prim als auch öffentlich bekannt ist.

Wie wir bereits angekündigt haben, gehen wir auf die Erzeugung verteilter Signaturen aufgrund der Notwendigkeit, hierbei zu interagieren, nicht weiter ein.

3.9 verteilte Schlüsselerzeugung

In dem von uns verwendeten Modell für Key-Sharing (siehe Abschnitt 3.2) besteht die größte Schwachstelle in der Existenz eines vertrauenswürdigen Gebers. Da dieser das Schlüsselpaar erzeugt, liegt es bis zur Verteilung auf die übrigen Teilnehmer an einem einzelnen Ort. Dem Geber muß zugetraut werden, seine Berechnungen sicher durchführen zu können und anschließend seine Kopie des privaten Schlüssels verläßlich zu vernichten. Neben der Gefährdung der Vertraulichkeit des Schlüssels hängt auch die Einsatzfähigkeit des Key-Sharing-Systems an der Korrektheit der Berechnungen durch den Geber. Beide Probleme treten in Verfahren, die ohne einen Geber auskommen nicht auf. Ohne im Detail auf

diese Verfahren eingehen zu wollen, geben wir hier Verweise auf andere Arbeiten, die sich mit der Thematik auseinandersetzen.

Mit der verteilten Erzeugung und Verwendung von RSA-Schlüsseln beschäftigen sich Miyazaki, Sakurai und Yung [MSY00]. Sie greifen dabei eine Arbeit von Boneh und Frankel[BF97] auf, in der gezeigt wird, wie mehrere Parteien einen RSA-Schlüssel gemeinsam (ohne Geber) erzeugen können. Das dort beschriebene Verfahren ist allerdings kein redundantes Key-Sharing-Verfahren, weshalb sie es mit dem in Abschnitt 3.5.1 geschilderten Verfahren von Frankel, Gemmell, MacKenzie und Yung [FGPY97] kombineren. Dieses Verfahren kam auch im Rahmen des gemeinsamen Projekts des Fachgebiets Theoretische Informatik mit der japanischen Telefongesellschaft NTT [ABF$^+$99] zur Implementierung eines verteilten Zeitstempeldienstes zum Einsatz.

Ein verteiltes System zur Erzeugung von Signaturen nach dem Signaturstandard DSS inklusive Schlüsselerzeugung beschreiben Gennaro, Jarecki, Krawczyk und Rabin [GJKR96]. Dieses System unterstützt auch die Möglichkeit der proaktiven Erneuerung der Anteile.

3.10 Anwendungen

Für die geschilderten Techniken lassen sich eine Reihe von Anwendungen nennen. In der Literatur häufig angeführt werden private Schlüssel, die keinen einzelnen Personen gehören, sondern Unternehmen zuzuordnen sind. In diesem Fall könnten zum Beispiel eine Gruppe führender Angestellter gemeinsam über einen Signaturschlüssel verfügen können sollen. Da diese Diplomarbeit im Rahmen des FlexiTrust-Projektes entstanden ist, welches sich mit der Entwicklung einer Trustcenter-Software zum Betrieb einer Zertifizierungs- und Registrierungsstelle beschäftigt, haben wir uns vor allem für Einsatzmöglichkeiten in einer CA interessiert. Die beiden wesentlichen Aufgabenbereiche sehen wir im Schutz des CA-Schlüssels (mit dem Zertifikate ausgestellt werden) und der Recovery-Schlüssel, mit denen Benutzerschlüssel wiederhergestellt werden können (falls die CA Benutzerschlüssel verwahrt).

Die konsequenteste Möglichkeit wäre sicherlich die Verteilung der von der CA durchgeführten Signaturen auf mehrere Rechner, die jeweils mit einem Teilschlüssel ausgestattet sein würden. Dies war allerdings in den Planungen für die FlexiTrust-Software nicht vorgesehen und hätte größere Änderungen zur Folge gehabt. Statt dessen sollte lediglich eine einfach anzubindende Möglichkeit geschaffen werden, um den CA-Schlüssel verteilt zu speichern (er würde zur Laufzeit dann zusammengesetzt). Es sollte auch möglich sein, ohne großen Aufwand alternativ den bisherigen oder einen dritten Weg zur Schlüsselspeicherung zu verwenden. Hierzu haben wir die nötige Secret-Sharing-Funktionalität in eine für FlexiTrust (und andere Java-Anwendungen) leicht zugängliche Form gebracht.

Jenseits des FlexiTrust-Projekts seien hier drei Projekte genannt, die Key-Sharing zur Erstellung verteilter Signaturen einsetzen, nämlich der verteilte Zeitstempeldienst, den der Fachbereich gemeinsam mit NTT entwickelt hat [ABF$^+$99], die von Boneh, Malkin und Wu beschriebene Integration in einen

SSL-Webserver [WMB99] und das COCA-Projekt der Cornell Universität [ZSvR00], das eine verteilte Online-CA implementiert.

Um eine möglichst sichere Verwahrung von Benutzerschlüsseln zu ermöglichen, werden diese von der CA verschlüsselt gespeichert (siehe hierzu auch Abschnitt 4.1). Die Public-Key-Kryptographie macht es möglich, daß die CA diese Daten selbst nicht mehr unbedingt entschlüsseln kann, sondern jemand anders dafür zuständig sein kann. Durch Key-Sharing kann die Aufgabe des Key-Recovery auch auf mehrere Personen verteilt werden. Dabei wird gleichzeitig die Sicherheit des dafür notwendigen Schlüssels erhöht.

Kapitel 4

Schlüsselverwahrung

Sollte es zu einer Serie terroristischer Greueltaten kommen, von denen die Sicherheitsbehörden zeigen können, daß sie durch Abhörmaßnahmen verhindert hätten werden können, dann werden Regierungen sehr schnell mehr Unterstützung für Key Escrow bekommen.

Simon Singh
in *The Code Book*

Ein Thema von vor allem politischer Brisanz sind Schlüsselverwahrung (*Key Escrow*) und Schlüsselwiederherstellung (*Key Recovery*) durch jemand anderen als den Schlüsselinhaber. Insbesondere ist umstritten, ob Strafverfolgungsbehörden die Möglichkeit erhalten sollen, Kopien von privaten Dechiffrier-Schlüsseln zu erhalten, so daß sie in die Lage versetzt werden, verschlüsselte (und damit eigentlich vertrauliche) Kommunikation abzuhören. Neben grundsätzlichen Einwänden von Verfechtern der Grundrechte auf informationelle Selbstbestimmung gibt es hier zahlreiche technische Bedenken, ob man durch die Einführung einer solchen Abhörmöglichkeit nicht das Kryptosystem insgesamt zu unsicher machen würde. Wir wollen die Argumente für und gegen Key Escrow in Abschnitt 4.1 wiedergeben und in den darauf folgenden Abschnitten darstellen, wie man es realisieren könnte.

4.1 Key Escrow

Es gibt unterschiedliche Motivationen dafür, einen Schlüssel wiederherstellen zu wollen.

- wenn der Besitzer des Schlüssels diesen verliert, kann er an ihn adressierte, verschlüsselte Nachrichten nicht mehr lesen. Es ist davon auszugehen, daß ein Trustcenter, in dem der Schlüssel hinterlegt wurde, diesen ausfallsicher speichern könnte. In diesem Fall wird der Schlüssel nur auf ausdrücklichen Wunsch des Besitzers wiederhergestellt. Es ist denkbar, daß für diesen Vorgang seine aktive Beteiligung notwendig ist, indem er etwa eine geheime Information beisteuert. Hierdurch kann sichergestellt werden, daß das Trustcenter den Schlüssel nicht ohne seine Genehmigung

verwenden kann. Andererseits ergibt sich das Problem, daß der Schlüsselbesitzer diese geheime Information aufbewahren muß, ein Problem, das sich nicht grundsätzlich davon unterscheidet, den Schlüssel aufzubewahren.

- wenn der Schlüssel innerhalb eines Unternehmens oder einer sonstigen Organisation zur Verschlüsselung von geschäftlichen Nachrichten eingesetzt wird, kann ein berechtigtes Interesse der Organisation bestehen, den Schlüssel wiederherstellen zu können, beispielsweise nachdem der damit betraute Mitarbeiter das Unternehmen verlassen hat. In diesem Fall kann etwa ein Vorgesetzter in die Lage versetzt werden, die (geschäftliche) Korrespondenz seiner Untergebenen einzusehen. Dies kann zwar auch ohne Beiteiligung und Wissen der eigentlichen Empfänger der Nachricht geschehen, aber üblicherweise gibt es keinen Grund, die betroffenen Mitarbeiter davon nicht in Kenntnis zu setzen.

- durch den Einsatz von starker Kryptographie stehen Strafverfolgungsbehörden vor dem Problem, daß Abhörmaßnahmen als Ermittlungswerkzeug nicht mehr eingesetzt werden können. In diesem, umstrittensten Einsatzgebiet von Key Recovery sollen Schlüssel ausdrücklich ohne Wissen und Zustimmung der Betroffenen wiederhergestellt werden und den Behörden zugänglich gemacht werden. Neben dieser Verdecktheitseigenschaft besteht noch ein weiterer Unterschied zu anderen Formen von Key Recovery, nämlich daß der Zugriff nicht auf gespeicherte Daten beschränkt bleibt, sondern (vor allem) auch Kommunikationsvorgänge entschlüsselt werden sollen.

Signaturschlüssel In all den oben genannten Situationen geht es um Dechiffrier-Schlüssel. Im Gegensatz dazu gibt es keinen Grund, Signaturschlüssel wiederherstellen zu wollen. Vielmehr muß dies verhindert werden: Wenn ein Signaturschlüssel verloren geht, können keine weiteren Signaturen ausgestellt werden. Trotzdem bleiben die bis dahin geleisten Signaturen gültig und können auch anhand des öffentlichen Schlüssels verifiziert werden. Um neue Signaturen ausstellen zu können, kann einfach ein neues Schlüsselpaar erzeugt werden, so daß sich kein großer Nutzen darin ergibt, den verlorenen Schlüssel wiederherstellen zu können. Die Unbequemlichkeit, ein neues Schlüsselpaar verwenden zu müssen, steht dem Risiko gegenüber, daß jemand anders den Schlüssel wiederherstellen und damit unberechtigt digitale Signaturen ausstellen könnte. Dies ist auf jeden Fall zu vermeiden.

Es gibt also (auch von seiten staatlicher Überwachungsorgane) keinen nachvollziehbaren Anspruch, ein Key Escrow für Signaturschlüssel zu betreiben, da der einzige Zweck eines solchen Unterfangens Urkundenfälschung sein könnte. Durch eine solche Maßnahme würde man die ökonomisch sehr bedeutsame Rechtswirksamkeit digitaler Signaturen gefährden. Im deutschen Signaturgesetz ist vorgeschrieben, daß digitale Signaturschlüssel nicht verwahrt oder wiederhergestellt werden dürfen. Außerdem dürfen diese Schlüssel nicht für andere Zwecke (zum Beispiel Verschlüsselungen) eingesetzt werden.

4.1.1 Key Recovery für Strafverfolgungsbehörden

Gegen ein umfassendes und verbindliches Key-Recovery-System, wie es noch vor wenigen Jahren intensiv diskutiert wurde, sind zahlreiche grundsätzliche und technische Bedenken geäußert worden. Vor allem durch die explosive Entwicklung des E-Commerce, der sich in seiner wachsenden wirtschaftlichen Bedeutung stark auf kryptographische Methoden stützt, haben sich seither die Befürworter einer unreglementierten Kryptographie durchsetzen können. Dies muß jedoch nicht für alle Zeiten so bleiben und deshalb wollen wir ihre bekanntesten Argumente skizzieren.

grundsätzliche Einwände Ein sehr fundamentaler Einwand bezieht sich auf eventuelle Verletzungen bürgerlicher Grundrechte durch ein Key-Recovery-System. Insbesondere wird mit dem Recht auf Privatsphäre, inklusive dem Recht auf vertrauliche Kommunikation, und dem Recht auf freie Meinungsäußerung argumentiert.

Desweiteren wird darauf hingewiesen, daß die Möglichkeit zum Key-Recovery mißbraucht und beispielsweise nicht nur im Kampf gegen das organisierte Verbrechen sondern auch gegen politische Gegner eingesetzt werden könnte. Nicht jeder vertraut seiner Regierung so weit, daß er ihr weitere Mittel zur Kontrolle der Bürger zur Verfügung stellen möchte. Eine nicht-staatliche oder gar private Organisation wird man mit der Aufgabe ebensowenig betrauen wollen.

In der letzten Zeit haben es die Regierungen aufgegeben, die Kontrolle über kryptographische Algorithmen behalten zu wollen. Gerade das Aufkommen frei verfügbarer Kryptosoftware macht jede Art von Reglementierung schwierig. Eine Umkehr hierbei müßte gegen den erbitterten Widerstand der Wirtschaft und der radikalen Verfechter freier Software, die jede Art von Zensur, Exportbeschränkung oder auch nur Patentierung ablehnen, durchgeführt werden, zudem auch noch auf internationaler Ebene. Und selbst wenn es gelänge, alle am Markt erhältlichen Kryptoprodukte zu reglementieren, darf man vermuten, daß gerade Kriminelle sich nicht an die Vorschriften bezüglich legaler Kryptographie halten und statt dessen illegale Produkte einsetzen, mit denen kein Key-Recovery möglich ist.

technische Einwände Es gibt auch eine Reihe Einwände technischer Natur, die gegen Key-Escrow-Systeme vorgebracht wurden. In einer gemeinsamen Erklärung [AAB+97] zu Key-Escrow zur Unterstützung der Arbeit von Strafverfolgungsbehörden haben Harold Abelson, Ross Anderson, Steven Bellovin, Josh Benaloh, Matt Blaze, Whitfield Diffie, John Gilmore, Peter Neumann, Ronald Rivest, Jeffrey Schiller und Bruce Schneier 1997 den technisch orientierten Teil der Diskussion zusammengefaßt. Wir wollen hier ihre wesentlichen Argumente wiedergeben.

- durch Möglichkeiten des Key Recovery werden zusätzliche Schwachstellen und Risiken in das Kryptosystem eingebracht. Die beiden größten Problem sind der Verlust der Garantie, daß es keinen anderen Zugang zu Klartexten gibt als den Benutzerschlüssel und die Tatsache, daß die

mit Key-Recovery beauftragten Institutionen neue, extrem attraktive Angriffsziele darstellen.

- ein solches System kann nur sinnvoll sein, wenn es umfassend, womöglich weltweit, eingesetzt wird. Aufgrund der dann notwendigen Zusammenarbeit zahlloser Behörden, Länder, Softwarehersteller und Benutzer zur Verwaltung der Millionen von Benutzer- und Sitzungsschlüsseln, die von Tausenden unterschiedlicher Kryptoprodukte teilweise ad hoc erzeugt werden, entstehen unbeherrschbare Komplexitäten. Insbesondere entsteht ein Widerspruch zwischen den von den Strafverfolgungsbehörden geforderten kurzen Bearbeitungszeiten und den organisatorischen Maßnahmen zur Sicherstellung der Prüfung der Legitimation eines Recovery-Antrags sowie dessen ordnungsgemäßer Durchführung.

- durch die Entwicklung, den Betrieb und die Überwachung eines Key-Recovery-Systems werden beträchtliche Kosten für Hersteller, Regierung und Benutzer entstehen

Zusammenfassend äußern sich die Autoren entschieden gegen jede Art von staatlich verordnetem Key-Escrow. Die Argumentation zielt im wesentlichen auf die Unbeherrschbarkeit einer umfassenden Key-Escrow-Lösung ab. Unabhängige und selbstbestimmte Systeme auf lokaler oder unternehmensweiter Ebene scheinen ihnen hingegen sinnvoll zu sein, zumal es dabei weniger um die Überwachung von Kommunikation als um den Schutz vor dem Verlust gespeicherter Informationen geht.

4.2 Schlüsselverwahrung

Die scheinbar einfachste Möglichkeit, Key-Recovery zu ermöglichen, besteht darin, Kopien von allen erzeugten privaten Schlüsseln aufzubewahren. Die Archivierung und Verwaltung der dabei entstehenden Daten ist eine extrem sicherheitskritische Angelegenheit und erfordert eine vertrauenswürdige Instanz. Das Vertrauen, das in eine solche Instanz gesetzt wird, hat dabei eine andere Qualität als das einer Zertifizierungsstelle entgegengebrachte, da es hierbei nicht darum geht, ob man die Instanz für fähig hält, Identitäten zu überprüfen, sondern ihr zutraut, einen privaten Schlüssel sicher aufzubewahren und nicht zu mißbrauchen. Eine normale CA ist nicht in der Lage, die vertrauliche Kommunikation ihrer Benutzer mitzulesen, eine Key-Recovery-Stelle hingegen schon. Insofern macht es Sinn, diese beiden Instanzen zu trennen. Andererseits bringt es organisatorische Vorteile und vermindert die Zahl an notwendigen vertrauenswürdigen Stellen, wenn man Zertifizierungsinstanzen zu Trustcentern mit solch erweiterter Funktion ausbaut.

Offensichtlich funktioniert Key-Recovery anhand von Backups nur, wenn diese Backups auch angelegt wurden. Dies macht die Kooperation des Schlüsselinhabers notwendig, der seinen Schlüssel entweder vom Trustcenter erzeugen lassen muß, einen speziellen Schlüsselgenerator verwenden muß, der das Recovery-Konzept unterstützt (etwa den vor einiger Zeit von den USA propagierten

Clipper-Chip), oder aktiv seinen Schlüssel zur Verwahrung vorlegen muß. Während dies kein Problem darstellt, wenn der Benutzer mit dem Backup einverstanden ist, wird eine obligatorische Schlüsselverwahrung schwierig durchzusetzen.

Das Archiv der Schlüsselkopien stellt ein immenses Sicherheitsrisiko dar und muß entsprechend geschützt werden. Um eine Verschlüsselung dieser Daten führt somit kein Weg herum. Dadurch entstehen Key-Recovery-Schlüssel, die von den Recovery-Operatoren verwendet werden, um Benutzerschlüssel aus dem Archiv zu holen. Diese sind ebenfalls extrem kritisch. Eine Möglichkeit, diese Master-Schlüssel zu sichern, ist durch das in Kapitel 3 geschilderte Key-Sharing gegeben. Durch Key-Sharing kann man gleichzeitig die Kontrolle über das Key-Recovery auf mehrere Instanzen oder Personen verteilen, was die Vertrauenswürdigkeit des Systems bei den Benutzern erhöht.

4.3 wiederherstellbare Schlüssel

Eine interessante Möglichkeit, private Schlüssel zurückzugewinnen erhält man durch das Einbetten dazu ausreichender Informationen in den öffentlichen Teil des Schlüssels. Hierzu verwendet man spezielle Schlüsselgeneratoren, die bei der Erzeugung der Schlüsselpaare dafür sorgen, daß Recovery-Operatoren (und nur diese) zu einem späteren Zeitpunkt aus dem öffentlichen Schlüssel den privaten errechnen können. Da öffentliche Schlüssel vielfach repliziert gespeichert werden, besteht erstens keine Notwendigkeit, ein zusätzliches Archiv zu führen, und zweitens kein Risiko diese Daten zu verlieren. Andererseits eröffnen die im folgenden geschilderten Verfahren neue Angriffswege auf die Sicherheit der Benutzerschlüssel, denn die eingebaute Hintertür für das Key-Recovery könnte mißbraucht werden. Die dazu notwendigen Daten sind in den öffentlichen Schlüsseln und dem Schlüsselgenerator enthalten und damit weniger gut geschützt als in einem Archiv eines Trustcenters (es wird natürlich weiterhin eine private Information der Recovery-Operatoren benötigt).

Wir wollen in diesem Abschnitt einige Möglichkeiten aufzeigen, wie Schlüsselgeneratoren gestaltet werden können, die scheinbar ganz normale Schlüsselpaare erzeugen, gleichzeitig aber eine Hintertür vorsehen, die es dem Hersteller des Generators ermöglicht, den privaten Schlüssel aus dem öffentlichen abzuleiten. Dabei ist es wichtig, daß durch eine genaue Analyse des Generators zwar festgestellt werden kann, daß er dieses automatische Key-Recovery unterstützt, aber es darf nicht möglich werden, dadurch an Informationen zu kommen, die es ermöglicht, das Recovery auch tatsächlich durchzuführen. Im allgemeinen wird der Generator einen öffentlichen (Backup-) Schlüssel enthalten aber nicht den dazugehörigen privaten (Recovery-) Schlüssel.

4.3.1 Kleptographie

Adam Young und Moti Yung bezeichnen die folgenden Techniken als Kleptographie und modellieren die in den Schlüsselgeneratoren eingebauten Hintertüren als SETUP (*secretly embedded trapdoor with universal protection*), von denen sie drei Kategorien unterscheiden [YY96, YY97]. Derjenige, der eine SETUP einrichtet, wird von ihnen als Angreifer bezeichnet.

SETUP Eine (reguäre) SETUP ist eine Modifikation eines Kryptosystems, so daß

die Eingaben mit den Spezifikationen des ursprünglichen Kryptosystems übereinstimmen,

das modifizierte Kryptosystem die öffentliche Verschlüsselungsfunktion des Angreifers enthält, nicht aber dessen private Entschlüsselungsfunktion,

die Ausgabe den ursprünglichen Spezifikationen entspricht, gleichzeitig aber verschlüsselte Informationen über den Benutzerschlüssel enthält, die für den Angreifer leicht zugänglich sind,

die Ausgaben der beiden Kryptosysteme ununterscheidbar sind (außer für den Angreifer) ,

es auch nach vollständiger Analyse des modifizierten Algorithmus nicht möglich wird, die erzeugten Schlüssel zu rekonstruieren (außer durch den Angreifer).

schwache SETUP Eine schwache SETUP unterscheidet sich von einer regulären dadurch, daß der Besitzer eines privaten Schlüssel in der Lage ist, die Ausgabe (sein Schlüsselpaar) von einer gewöhnlichen Ausgabe zu unterscheiden. Bei einer regulären SETUP vermag dies nur der Angreifer.

starke SETUP Eine Kerneigenschaft der regulären SETUP ist die Ununterscheidbarkeit ihrer Ausgaben von den Ausgaben des unmodifizierten Kryptosystems. Nach vollständiger Analyse können die Benutzer des Algorithmus zwar aus seinen Ausgaben die darin enthaltenen sublimen Informationen nicht nutzen, wissen jedoch um deren Existenz. Eine starke SETUP nutzt die eingebaute Hintertür nicht immer, sondern greift gelegentlich auf das normale Verfahren zurück. Dabei bleiben die kontaminierten Schlüssel und die nichtkontaminierten Schlüssel ununterscheidbar.

Einsatzgebiet Als Anwendung einer SETUP nennen Young und Yung die hier vorgeschlagenen wiederherstellbaren Schlüssel (*auto-escrowing keys*). Zugleich stehen sie dem Einsatz eines solches Systems kritisch gegenüber, was sich nicht zuletzt in der Namensgebung für ihr System (PAP – Pretty Awful Privacy) ausdrückt. Auch wir möchten durch unsere Implementierung der kleptographischen Algorithmen nicht deren tatsächlichen Einsatz in einer PKI empfehlen.

4.3.2 RSA-Schlüssel

Bei der Erzeugung eines RSA-Schlüsselpaares werden zwei zufällige Primzahlen p und q gewählt, deren Produkt $N = pq$ den RSA-Modulus ergibt, sowie zwei Exponenten d und e mit $de = 1 \pmod{(p-1)(q-1)}$ (für Details des RSA-Verfahrens siehe Abschnitt 3.3). Die Primzahlen p und q werden im Laufe

des Verfahrens nicht mehr benötigt (außer um Berechnungen mit dem privaten Schlüssel zu beschleunigen) und müssen nicht gespeichert werden. Da die Kenntnis der Faktorisierung von N ausreichend ist, um d aus e zu berechnen, sind p und q auf jeden Fall geheim zu halten. Die hier geschilderten Verfahren bringen eine verschlüsselte Version von p in den öffentlichen Schlüssel ein, so daß der Recovery-Operator daraus den Primfaktor p (und damit auch $q = \frac{N}{p}$) entschlüsseln und den privaten Exponenten d berechnen kann.

Verschlüsselung im Exponenten e Das einfachere der beiden hier geschilderten Verfahren erzeugt p und q genauso wie ein normaler RSA-Schlüsselgenerator. Der Algorithmus enthält den öffentlichen Backup-Schlüssel des Recovery-Operators. Mithilfe dieses Schlüssels verschlüsselt er den Faktor p und erhält einen Chiffretext c. Um die Sicherheit des Verfahrens nicht von anderen Algorithmen abhängig zu machen, sollte es sich bei dem Backup-Schlüssel ebenfalls um einen RSA-Schlüssel handeln. Falls die Verschlüsselung c teilerfremd zu $(p-1)(q-1)$ ist, setzt der Generator den öffentlichen Exponenten $e = c$. Falls nicht, verschlüsselt er den anderen Faktor q, wenn dies ebenfalls nicht zum Erfolg führt, wählt er solange neue Primfaktoren p, bis sich ein geeignetes c ergibt. Der private Exponent d errechnet sich aus e, p und q mit dem erweiterten Euklidschen Algorithmus.

Um den Schlüssel wiederherzustellen, betrachtet der Recovery-Operator den öffentlichen Exponenten e als Schlüsseltext und entschlüsselt ihn zum Primfaktor p. Wegen $N = pq$ erhält er damit zugleich auch den anderen Faktor q und kann mit dem erweiterten Euklidschen Algorithmus den privaten Exponenten d als multiplikatives Inverses zu e modulo $(p-1)(q-1)$ bestimmen.

Das Problem bei dieser Vorgehensweise liegt daran, daß es nicht immer möglich ist, den öffentlichen Exponenten e frei zu wählen. Viele RSA-Implementierungen verlangen nach einem kleinen e, weil sich dies positiv auf die Rechenzeit der öffentlichen RSA-Operationen auswirkt ohne einen negativen Effekt auf die Sicherheit zu haben. Das obige Verfahren erzeugt jedoch Exponenten e mit einer Bitlänge die der verwendeten Backup-Chiffre und damit (falls RSA verwendet wurde) aus Sicherheitsüberlegungen der Bitlänge des Moduln N, mindestens jedoch der Bitlänge von p entspricht.

Verschlüsselung im Moduln N Wenn öffentliche Exponenten e beliebiger Größe nicht möglich sind, kann der Chiffretext für den Primfaktor p auch in der oberen Hälfte des Moduln N untergebracht werden. Hierbei ist jedoch zu beachten, daß der Chiffretext dann auf die Hälfte der Länge von N beschränkt ist. Wenn man hierfür ebenfalls RSA einsetzen will, muß man Benutzerschlüssel erzeugen, die doppelt so lang sind wie der Recovery-Schlüssel. Dies ist sicher eine etwas ungünstige Situation.

Im folgenden gehen wir davon aus, daß der Recovery-Schlüssel ein RSA-Schlüssel mit Modul N_R und Exponenten d_R und e_R ist. Der Generator erzeugt zunächst die zufällige Primzahl p. Diese Primzahl p wird anschließend durch eine symmetrische Chiffre F mit festem Schlüssel K in eine Zahl p' überführt. Der Schlüssel K ist dabei fest in den Generator integriert. Er muß nicht ge-

heimgehalten werden, da die Chiffre F nur der Randomisierung von p' gilt. Falls p' kein gültiger RSA-Klartext für die Recovery-Chiffre ist (es muß gelten $p' < N_R$), so wird der Schlüssel K um eins weitergezählt ($K \leftarrow K + 1$) und p' wird erneut berechnet. Diese zweite Funktion der Chiffre F vermeidet das häufige Suchen nach geeigneten Primzahlen p und senkt so die Rechenzeit. Erst nach einer bestimmten Anzahl an Iterationen B_1 wird abgebrochen und das Verfahren neu gestartet (mit der Suche nach einem neuen p).

Sobald ein gültiges p' gefunden wurde, wird dieses mit dem Backup-Schlüssel verschlüsselt. Von dem entstehenden Schlüsseltext $c = (p')^{e_R} \pmod{N_R}$ wird nun erneut durch die symmetrische Chiffre F mit anfänglichem Schlüssel K eine Reihe von Kandidaten c' für die obere Hälfte von N erzeugt (jeweils durch Weiterzählen von K). Testweise wird jeder Kandidat c' durch Konkatenation mit zufälligen Bits auf die doppelte Länge gebracht. Die entstehende Zahl wird durch p geteilt (Ganzzahldivision ohne Rest), falls dabei eine Primzahl q entsteht, setzt man $N = pq$ und hat damit einen RSA-Modulus erhalten, dessen obere Hälfte aus dem Chiffretext c' besteht (aufgrund von Übertragsbits und dem fehlenden Rest bei der Ganzzahldivision kann die obere Hälfte $N_L = c'(-1)$ auch um eine Einheit zu niedrig sein). Hierzu werden mehrere Versuche notwendig sein, allerdings aufgrund des Primzahltheorems nicht allzu viele. Die Chiffre F erzeugt ohne großen Rechnenaufwand eine gewisse Anzahl randomisierter Kandidaten. Auch hier muß das Verfahren erst bei einer vorgegebenen Iterationsschranke B_2 neu gestartet werden (mit einem neuen p).

Zur Verdeutlichung der Funktionsweise der Suche nach N wollen wir ein kleines und stark vereinfachtes Beispiel geben. Wir wollen einen RSA-Schlüssel mit 6 Dezimalstellen erzeugen, wobei der Primfaktor p den umgekehrt gelesenen ersten drei Ziffern des Moduln N entspricht. Zufällig entnehmen wir aus der Primzahltabelle die Kandidaten $p \in \{271, 953, 647, 443, 751, 337, 607, 823, 523, 107\}$ und „verschlüsseln“ sie zu $c' \in \{172, 359, 746, 344, 157, 733, 706, 328, 325, 701\}$. Unser erster Versuch führt zu 172512 (die letzten drei Ziffern zufällig ergänzt), ganzzahlige Division durch 271 ergibt 636, keine Primzahl. Aber schon im dritten Versuch gelangen wir zu 746357, also $p = 647$, $q = 1153$ und $N = 745991$ (wobei der angesprochene Übertragsfehler aufgetreten ist). Von den zehn Kandidaten führten drei zum Erfolg (davon zwei mit Übertragsfehler).

Wenn auf diese Weise passende N, p und q gefunden wurden, kann man den öffentlichen Exponenten e frei wählen (solange $\mathrm{ggT}(e, (p-1)(q-1)) = 1$ gilt) und den dazugehörigen privaten Exponenten d mit dem erweiterten Euklidschen Algorithmus berechnen.

Um den Schlüssel wiederherzustellen, betrachtet der Recovery-Operator die obere Hälfte des Moduln N als Schlüsseltext c' der Chiffre F. Da er nicht genau wissen kann, welcher der Kandidaten c' zum Erfolg geführt hat (der symmetrische Schlüssel K wurde dabei jeweils weitergezählt), entschlüsselt er c' für jedes mögliche K (deren Zahl durch die Iterationsschranke B_2 begrenzt ist) und erhält damit eine Reihe denkbarer RSA-Schlüsseltexte c, die er mit seinem Recovery-Schlüssel zu einer Menge von möglichen p' entschlüsseln kann. Auch hier muß er wieder die symmetrische Chiffre F mit den B_1 möglichen Schlüsseln K anwenden, solange bis er einen Primfaktor p von N entschlüsseln konnte. Konnte kein p gefunden werden, muß er aufgrund des möglichen Übert-

ragsfehlers die Berechnungen auch noch für die um eine Einheit erhöhte obere Hälfte von N durchführen.

4.3.3 ElGamal-Schlüssel

Bei der Erzeugung eines ElGamal-Schlüssels (p, g, a, A) wollen wir davon ausgehen, daß alle Parameter vom Schlüsselgenerator frei bestimmt werden können. Dadurch wird es möglich, den geheimen Exponenten a mit einem ElGamal-Backup-Schlüssel zu (B, c) zu verschlüsseln und diese beiden Werte in den Parametern p und g zu verstecken. Young und Yung [YY96] geben darüber hinaus auch zwei Algorithmen an, die mit einem vorgegebenen p oder einem vorgegebenen g arbeiten können.

Der Schlüsselgenerator enthält den Backup-Schlüssel (p_R, g_R, A_R). Als erstes erzeugt er den zufälligen Exponenten a. Dieser Wert wird mit dem Backup-Schlüssel und einem zufälligen k zu $(B, c) = (g_R^k, A_R^k a) \pmod{p_R}$ verschlüsselt. Falls möglich, setzt er die Parameter p und g des zu erzeugenden Schlüssels als $p \leftarrow c$ und $g \leftarrow B$. Hierzu muß c eine Primzahl sein ($c-1$ sollte außerdem einen großen Primfaktor haben) uns es muß gelten $a < c$ und $B < c$. Ist dies nicht der Fall, wird die Verschlüsselung mit einem neuen k wiederholt (Young und Yung fordern nicht, daß g tatsächlich die gesamte Gruppe $(\mathbb{Z}/p\mathbb{Z})^*$erzeugt, obwohl dies im ElGamal-Verfahren eigentlich vorgesehen ist). Der letzte Parameter $A = g^a \pmod{p}$ ergibt sich abschließend aus p, g und a.

Zum Key-Recovery werden die Parameter g und p des öffentlichen Schlüssels einfach als ElGamal-Chiffretext aufgefaßt und mit dem Recovery-Schlüssel zu a entschlüsselt.

Kapitel 5

Implementierung

Der Worte sind genug wechselt, laßt mich auch endlich Taten sehen.

aus Goethes *Faust*

Einen Teil der vorgestellten Verfahren aus den Bereichen Secret-Sharing, Key-Sharing und Key Escrow haben wir in der Programmiersprache Java implementiert. Die Implementierung war dabei auf die Erfordernisse im Rahmen des FlexiTrust-Projekts des Fachgebiets ausgerichtet. In diesem Projekt wird eine Trustcenter-Software entwickelt, die durch diese Diplomarbeit um verteilt gespeicherte CA-Schlüssel und einen Key-Recovery-Mechanismus für Benutzerschlüssel erweitert werden sollte.

Die Dokumentation der Implementierung besteht aus drei Teilen:

- Einen Überblick über die Implementierung stellt dieses Kapitel der Diplomarbeit dar. Hier werden die einzelnen Teile, ihr Zusammenspiel, sowie die bei der Programmierung getroffenen Entscheidungen erläutert. UML-Klassendiagramme sollen den Zugang erleichtern.

- In Java integriert ist eine Dokumentationsschnittstelle für Klassenbibliotheken namens JavaDoc. Hiermit lassen sich Java-Klassen für die Verwendung in anderen Programmen dokumentieren. Die von JavaDoc erzeugten (sehr ansehnlichen) HTML-Dateien dienen damit im wesentlichen dem Anwendungsprogrammierer.

- JavaDoc beschreibt die Schnittstelle einer Klasse wie eine Black Box. Interne Details der Programmierung werden nicht sichtbar gemacht. Hierzu kann man auf den Quellcode und die dort enthaltenen Kommentare zurückgreifen, der sich im CVS-Repositorium des FlexiTrust-Projektes befindet.

5.1 Kryptographie in Java

Ein fester Bestandteil der Java-Laufzeitumgebung ist die Java Cryptography Architecture (JCA), die dem Programmierer eine umfangreiche Schnittstelle zum Zugriff auf kryprographische Funktionen bietet. Dabei ist es vorgesehen,

daß die einzelnen Algorithmen von verschiedenen Anbietern (Provider) implementiert werden können, die Verwaltung der verschiedenen (auch gleichzeitig installierten) Provider für den Anwender jedoch transparent, auf dessen Wunsch aber auch zur Laufzeit beeinflußbar ist. Da wir Teile unserer Implementierung über JCA anbieten, ist ein Grundverständnis der Konzepte von JCA notwendig, so daß wir an dieser Stelle einen kurzen Überblick geben wollen. Zur Vertiefung sei auf die Literatur [Oak98] und die Internetseiten von Sun Microsystems verwiesen.

Ein Teil der JCA-Klassen wird auch als Java Cryptography Extension (JCE) bezeichnet. Im Gegensatz zu den übrigen JCA-Klassen, die sich in den `java.security`-Paketen befinden, liegen die JCE-Klassen in `javax.crypto`-Paketen. Diese Aufteilung war aufgrund von mittlerweile gelockerten Exportvorschriften der US-Regierung notwendig, ist dann aber hinfällig geworden, so daß auch JCE inzwischen zum Standardumfang der Laufzeitumgebung gehört. Wir werden im folgenden nicht zwischen JCA und JCE unterscheiden.

5.1.1 Schlüssel

JCA modelliert private, öffentliche und symmetrische Schlüssel und bietet Klassen zu deren Erzeugung, Konvertierung und Speicherung an.

`Key` diese Schnittstelle gruppiert alle kryptographischen Schlüssel. Sie schreibt lediglich vor, daß ein Schlüssel einem Algorithmus zugeordnet wird und eine Binärkodierung unterstützen sollte.

`PrivateKey` diese Schnittstelle ohne Methoden dient nur zur typsicheren Unterscheidung von öffentlichen und privaten Schlüsseln.

`PublicKey` diese Schnittstelle ohne Methoden dient nur zur typsicheren Unterscheidung von öffentlichen und privaten Schlüsseln.

`KeyPair` diese Klasse faßt einen `PrivateKey` und einen `PublicKey` zu einem Schlüsselpaar zusammen.

`SecretKey` diese Schnittstelle ohne Methoden dient nur zur typsicheren Identifikation eines symmetrischen Schlüssels.

`KeySpec` zusätzlich zu den `Key`-Klassen gibt es noch die `KeySpec`-Klassen, die beschreibende Informationen über einen Schlüssel enthalten, anhand derer eine `Key`-Instanz erzeugt werden kann. Dabei kann es sich um binäre Beschreibungen (wie PKCS8) handeln oder auch um algorithmenabhängige offene Beschreibungen, etwa zwei `BigInteger`-Werte e und N zur Spezifikation eines öffentlichen RSA-Schlüssels.

`KeyPairGenerator` ein `KeyPairGenerator` erzeugt ein neues Schlüsselpaar (`KeyPair`).

`KeyFactory` eine `KeyFactory` transformiert Schlüsselbeschreibungen (etwa Binärformate) in `Key`-Instanzen.

`KeyStore` der `KeyStore` eignet sich zur sicheren Aufbewahrung kryptographischer Schlüssel. Die Integrität des Speichers sowie jeder einzelne Schlüssel werden durch Paßworte gesichert. Die Art der Speicherung ist je nach Implementierung unterschiedlich. Sun liefert eine proprietäte dateibasierte Implementierung, den Java KeyStore (JKS). Andere Möglichkeiten sind PKCS12 oder Chipkarten.

5.1.2 Zertifikate

Um Public-Key-Kryptographie sinnvoll betreiben zu können, ist der Umgang mit Zertifikaten unerläßlich. Diese beinhaltet die Möglichkeit, die binär kodierten Zertifikate einlesen und auswerten zu können, also öffentliche Schlüssel zu entnehmen, Signaturen zu prüfen, Aussteller zu identifizieren und Zertifikatsinhalte (Erweiterungen und Einschränkungen) zu behandeln.

`Certificate` diese Basisklasse stellt ein Public-Key-Zertifikat dar.

`X509Certificate` als Erweiterung von `Certificate` handelt es sich hierbei um ein Zertifikat nach dem X.509-Standard. Es gibt unter anderem Methoden um den Inhaber und seinen öffentlichen Schlüssel, den Aussteller oder den Gültigkeitszeitraum zu erfahren, sowie um die Signatur zu prüfen (hierzu muß der öffentliche Schlüssel des Ausstellers vorliegen).

`CRL` diese Klasse repräsentiert eine Revokationsliste (*certificate revocation list*). Hier sind Zertifikate aufgeführt, die für ungültig erklärt wurden. Es gibt auch eine Unterklasse für X.509-Revokationslisten.

`CertificateFactory` eine `CertificateFactory` erzeugt `Certificate`-Instanzen aus Eingabeströmen, zum Beispiel aus Dateien.

`KeyStore` neben der Speicherung von Schlüsseln dient ein `KeyStore` auch dazu, vertrauenswürdige Zertifikate abzulegen, anhand derer Signaturen auf Benutzerzertifikaten überprüft werden sollen. Durch das Integritätspaßwort wird sichergestellt, daß nicht unbemerkt Zertifikate entfernt oder installiert werden können.

5.1.3 Algorithmen

Für die verschiedenen kryptographischen Operationen sind jeweils eigene Klassen vorgesehen, die vom Programmierer mit einer `getInstance()`-Methode aktiviert werden können. Die Implementierung wird dabei erst zur Laufzeit ausgewählt und dynamisch geladen. Auf diese Weise wird eine Java-Anwendung von den unterschiedlichen Algorithmen und den Bibliotheken, in denen diese bereit gestellt werden, entkoppelt. Beispielsweise wird eine MD5withRSA-Signatur gemäß JCA wie folgt erzeugt:

```
Signature signer = Signature.getInstance("MD5withRSA");
signer.initSign((PrivateKey)privateKey);
signer.update((byte[]) message);
byte[] signature = signer.sign();
```

Um den Signaturalgorithmus auszutauschen, muß lediglich ein anderer Algorithmenname angegeben werden. Kann kein Algorithmus mit diesem Namen gefunden werden, hat dies eine `NoSuchAlgorithmException` zur Folge. Natürlich müssen die verwendeten Schlüssel zu den Algorithmen passen. Ist dies nicht der Fall, wird ebenfalls eine entsprechene Exception ausgelöst.

Mit JCA ist es möglich, mehrere Kryptographie-Bibliotheken (sogenannte Service-Provider) nebeneinander zu verwenden. An welchen der installierten Provider der Anruf von `getInstance()` weitergereicht wird entscheidet JCA zur Laufzeit anhand einer konfigurierbaren Priorisierung und des Algorithmenangebots der Provider. Es ist ferner möglich, explizit einen Providernamen anzugeben, der verwendet werden soll. Falls dieser nicht existiert oder den gewünschten Algorithmus nicht anbietet, werden eine `NoSuchProviderException` oder `NoSuchAlgorithmException` geworfen.

`Signature` diese Klasse wird zum Erzeugen und Prüfen digitaler Signaturen verwendet. Sie wird mit einem geeigneten Schlüssel zum Signieren oder Verifizieren initialisiert. Die Nachricht wird mit einer `update(byte[]`-Methode übergeben, die Signatur mit `sign()` erzeugt oder mit `verify(byte[] signature)` überpüft.

`Cipher` diese Klasse eignet sich zur Ver- und Entschlüsselung von Daten. Je nach Algorithmus wird sie mit öffentlichen oder symmetrischen Schlüsseln zur Verschlüsselung, mit privaten oder symmetrischen Schlüsseln zur Entschlüsselung initialisiert. Auch hier gibt es wieder `update(byte[])`-Methoden, das Ergebnis entsteht durch den Aufruf von `doFinal()`.

`MessageDigest` diese Klasse erzeugt kryptographische Hashwerte (Digests, Fingerprints) von Nachrichten. Hashwerte werden verwendet um die Integrität einer Nachricht zu überprüfen (wobei zumindest der Hashwert aus sicherer Quelle stammen muß).

`Mac` diese Klasse implementiert *Message Authentication Codes*. Hierbei handelt es sich um integritätssicherende Prüfsummen, die auf symmetrischen Schlüsseln basieren (im Gegensatz zu Signaturen, die auf asymmetrischer Kryptographie aufbauen, und zu reinen Hashwerten, die überhaupt nicht geschützt sind).

`SecureRandom` diese Klasse erzeugt kryptographisch sichere (also nicht vorhersehbare) Pseudozufallszahlen.

`AlgorithmParameterSpec` viele Algorithmen müssen parametrisiert werden. Hierzu existieren die `AlgorithmParameterSpec`-Klassen, die je nach Algorithmus ganz unterschiedlich Gestalt annehmen können. Durch diese gemeinsame Schnittstelle können sie von der JCA zumindest durchgereicht werden.

5.1.4 Schnittstellen für Dienstanbieter

Die in den vorherigen Abschnitten geschilderten Klassen stellen die Schnittstelle für den Anwendungsprogrammierer dar. Spiegelbildlich dazu gibt es eine

entsprechende Schnittstelle für Dienstanbieter, die eigene Implementierungen liefern wollen. Diese Schnittstelle besteht im wesentlichen aus Service-Provider-Interface-Klassen für die jeweiligen Konzepte, zum Beispiel `SignatureSpi` oder `KeyStoreSpi`. Um einen JCA-konformen Algorithmus zu implementieren, muß man diese abstrakten Basisklassen geeignet erweitern und die Klasse im Provider registieren.

Wie wir gesehen haben, lädt JCA eine implementierende Klasse automatisch anhand des Algorithmen- und des (optionalen) Providernamens, die einer `getInstance(String, String)`-Methode übergeben wurden. Damit dies funktioniert müssen die Dienstanbieter ihre Implementierungen registrieren. Hierfür existiert die Klasse `Provider`. Im Prinzip handelt es sich dabei um eine Tabelle, die einem Algorithmennamen den Namen der implementierenden Klasse zuordnet, die dann dynamisch geladen werden kann. Die Providerklassen selbst werden dem Laufzeitsystem entweder über Systemeigenschaften oder zur Laufzeit durch `Security.addProvider(Provider)` bekannt gemacht.

5.1.5 ASN1-Codec

Ein Aspekt der durch JCA nicht abgedeckt wird ist die Erzeugung von ASN1-Datenstrukturen. Gemäß diesem Kodierungsstandard werden beispielsweise X.509-Zertifikate geschrieben. Infolgedessen kann man mit JCA zwar Zertifikate lesen und auswerten, aber nicht ausstellen. Wir haben daher ein ursprünglich von der Fraunhofer-Gesellschaft entwickeltes ASN1-Codec verwendet, welches mittlerweile auf den Fachbereich Theoretische Informatik übergegangen ist.

5.2 Überblick über die Java-Klassen

Die Java-Klassen sind in eine Reihe von Paketen gruppiert:

`keyshare` das Paket enthält die grundlegenden Algorithmen und Datenstrukturen zur Implementierung der Secret-Sharing- und Key-Sharing-Verfahren.

`keyshare.jca` das Paket enthält Hüllen- und Adapterklassen mit denen eine möglichst große Kompatibilität zur Java Cryptographic Architecture hergestellt werden soll. Das Paket enthält Implementierungen von `Cipher`, `Signature` und `KeyFactory` zum Umgang mit Schlüsselanteilen.

`keyshare.keystore` das Paket enthält verschiedene `KeyStore`-Implementierungen, die von Secret- und Key-Sharing Gebrauch machen. Zwei Unterpakete `keyshare.keystore.gui` und `keyshare.keystore.xml` stellen eine graphische Benutzerschnittstelle und Zugriff auf XML-Konfigurationsdateien bereit.

`keyshare.escrow` das Paket enthält Klassen zum Erzeugen und Wiederherstellen von wiederherstellbaren RSA- und ElGamal-Schlüsseln (siehe Abschnitt 4.3).

`keyshare.apps` das Paket enthält ausführbare Java-Kommandozeilen-Applikationen mit denen Secret- und Key-Sharing verwendet werden kann.

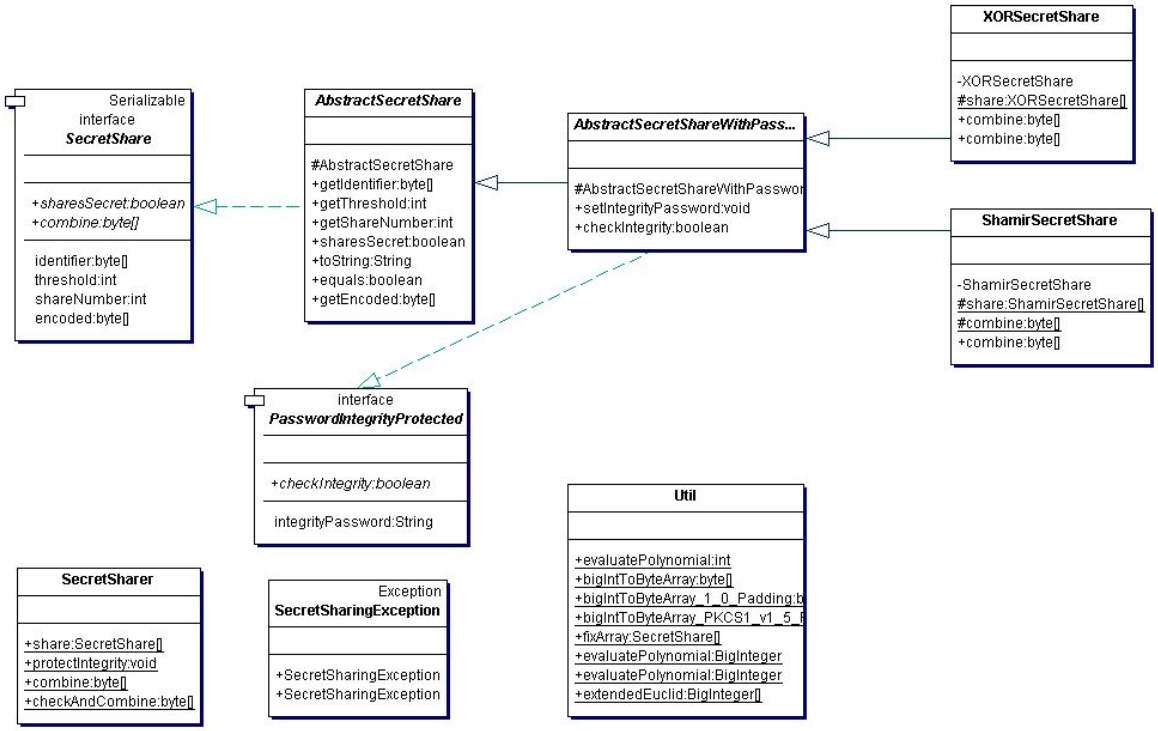


Abbildung 5.1: Secret-Sharing-Basisklassen aus dem Paket `keyshare`

`keyshare.tests` das Paket enthält Testfälle gemäß dem JUnit-Framework mit denen die korrekte Funktionsweise der Implementierung überprüft werden kann.

5.2.1 Paket `keyshare`

Das Paket `keyshare` enthält die grundlegenden Algorithmen und Datenstrukturen zur Implementierung sowohl der Secret-Sharing- als auch der Key-Sharing-Verfahren.

Secret-Sharing Es wurden das XOR-Secret-Sharing-Verfahren (Abschnitt 2.1) und das Shamir-Verfahren (Abschnitt 2.2.1) implementiert. Für beide Verfahren ist vorgesehen, daß die Anteile bei ihrer Erzeugung mit einem Integritätspaßwort geschützt werden können. Dies soll verhindern, daß Veränderungen der Anteile unerkannt bleiben. Nicht implementiert wurde das verifizierbare Pedersen-Verfahren (Abschnitt 2.3.1). Die Struktur der im folgenden kurz beschriebenen Klassen und Schnittstellen ist in Abbildung 5.1 dargestellt.

`SecretShare` diese Schnittstelle definiert die Methodensignatur eines Geheimnisanteils. Ein `SecretShare` muß Auskunft über seine Nummer, die Anzahl der benötigten Anteile, eine ID und darüber, ob er zu einem bestimmten Geheimnis paßt, geben. Außerdem muß es sich in ein Byte-Array se-

rialisieren lassen und die Methode zur Rekonstruktion des Geheimnisses implementieren (der man dann genügend viele Anteile übergeben muß).

`SecretSharingException` diese Ausnahme wird geworfen, wenn ein Fehler bei der Rekonstruktion eines Geheimnisses auftritt.

`PasswordIntegrityProtected` diese Schnittstelle wird von Geheimnisanteilen implementiert, die mit einem Integritätspaßwort geschützt werden sollen. Sie definiert Methoden zum Setzen und Überprüfen des Paßworts.

`AbstractSecretShare` diese abstrakte Klasse stellt Default-Implementierungen für die Methoden eines `SecretShare` bereit und dient damit zur Vermeidung von Code-Duplikaten.

`AbstractSecretShareWithPassword` diese abstrakte Klasse erweitert `AbstractSecretShare` um Unterstützung für die Schnittstelle `PasswordIntegrityProtected` und damit um ein Integritätspaßwort.

`ShamirSecretShare` diese Klasse stellt einen durch das Shamir-Verfahren entstandenen Anteil dar. Sie enthält auch Methoden zum Verteilen eines geheimen Byte-Arrays nach diesem Verfahren.

`XORSecretShare` diese Klasse stellt einen durch das XOR-Verfahren entstandenen Anteil dar. Sie enthält auch Methoden zum Verteilen eines geheimen Byte-Arrays nach diesem Verfahren.

`SecretSharer` diese Klasse enthält statische Methoden zum einfachen Zugriff auf die Secret-Sharing-Funktionalität (Erzeugen, Prüfen und Kombinieren von Anteilen). Sie bietet damit eine Fassade für den Secret-Sharing-Teil des Pakets.

`Util` diese Klasse enthält einige statische Hilfsmethoden, die von anderen Klassen verwendet werden können.

blockweises Shamir-Secret-Sharing In der Schilderung des Shamir-Verfahrens in Abschnitt 2.2.1 sind wir davon ausgegangen, daß sich das Geheimnis einfach als Zahl auffassen und durch ein Polynom verteilen läßt. Dies ist bei Geheimnissen, die mehr als nur ein paar Byte lang sind, zwar möglich, aber unpraktikabel. Sinnvoller ist es, das Geheimnis in mehrere Blöcke aufzuteilen und diese einzeln zu verteilen. Dabei ergibt sich für jeden Block ein neues Polynom. Die Blockgröße kann beliebig gewählt werden, so daß man sich bei den Berechnungen auf Zahlenbereiche beschränken kann, die vom Computer effizient bearbeitet werden können. In unserer Implementierung haben wir das Geheimnis byteweise verteilt. Durch vorgeschaltete Base64-Kodierung bestanden die Bytes des Geheimnisses nur aus druckbaren ASCII-Zeichen, so daß wir Polynome modulo 127 verwenden konnten. Durch den festen Moduln konnten wir auch die notwendigen Invertierungen als Vorberechnungen ausführen und in einer Tabelle in das Programm integrieren.

MD5 zum Berechnen der ID Jedes `SecretShare` verfügt über eine ID, die es ermöglicht, zusammengehörige Sätze von Anteilen zu erkennen. Diese ID ergibt sich durch einen Hashwert über das jeweilige Geheimnis und wird beim Erzeugen der Anteile (dem Aufteilen des Geheimnisses) berechnet. Wir haben uns für die MD5-Hashfunktion entschieden, die in jeder Laufzeitumgebung per JCA verfügbar sein sollte. Die ID eines Anteils ist somit 16 Byte lang.

Paßwort-Integritätsschutz Wir haben das Pedersen-Verfahren für verifizierbares Secret-Sharing nicht implementiert, weil es nicht ganz in unser Modell paßt und der Schutz von fehlerhaft berechneten Anteilen auch nicht benötigt wird (wir vertrauen dem Geber). Um trotzdem dem Kombinierer die Möglichkeit zu geben, von den Teilnehmern verfälschte Anteile zu erkennen, haben wir die Möglichkeit vorgesehen, daß der Geber die Anteile mit einem (den Teilnehmern unbekannten) Paßwort versieht. Dieses Paßwort geht dann gemeinsam mit der ID und dem Inhalt des Anteils in eine Prüfsumme ein (wieder MD5), die ohne Kenntnis des Paßworts nicht manipuliert werden kann. Auf diese Weise können bei der Kombination die Beiträge der Teilnehmer anhand des Paßwortes überprüft werden.

Double Dispatch Das Interface `SecretShare` schreibt eine Methode `combine()` vor, mit der eine Menge gleichartiger `SecretShare`-Instanzen zusammengesetzt werden können. Diese Methode ist in allen konkreten Implementierungen von `SecretShare` derart realisiert, daß das hereingereichte Feld von `SecretShare[]` auf den konkreten Typ (beispielsweise `ShamirSecretShare[]`) eingeengt wird und dann einer weiteren `combine()`-Methode weitergereicht wird, welche die tatsächlichen Berechnungen durchführt. Dieser explizite Typecast ist notwendig, da Java beim Aufruf einer Methode lediglich den Typ des Empfängerobjekts dynamisch ermittelt, nicht jedoch die Typen der Argumente. Man bezeichnet diese Vorgehensweise als Single Dispatch. Würde Java direkt einen Double Dispatch unterstützen, hätten wir die Adaptermethoden `combine()`, in denen lediglich ein Typecast vorgenommen wird, nicht in jeder Klasse implementieren müssen.

Key-Sharing Es wurden die einfachen (nicht redundanten) Key-Sharing-Verfahren für RSA (Abschnitt 3.4) und ElGamal (Abschnitt 3.8), die beiden redundaten RSA-Key-Sharing-Verfahren von Frankel, Gemmell, MacKenzie und Yung (in der abgewandelten Version aus Abschnitt 3.5.2) sowie von Shoup (Abschnitt 3.6) und das redundante Key-Sharing-Verfahren für spezielle ElGamal-Schlüssel (Abschnitt 3.8) implementiert. Die RSA-Teilschlüssel unterstützen sowohl Entschlüsselungen als auch das Erstellen von Signaturen, die ElGamal-Teilschlüssel eignen sich nur zum Entschlüsseln (aufgrund des angesprochenen Kommunikationsbedarfs bei Signaturen, siehe Abschnitt 3.8). Die Struktur der im folgenden kurz beschriebenen Klassen und Schnittstellen ist in Abbildung 5.2 dargestellt.

`KeyShare` diese Schnittstelle erweitert `SecretShare` um eine Methode zum Erfragen des Namen des Algorithmus, für den der verteilte Schlüssel geeignet

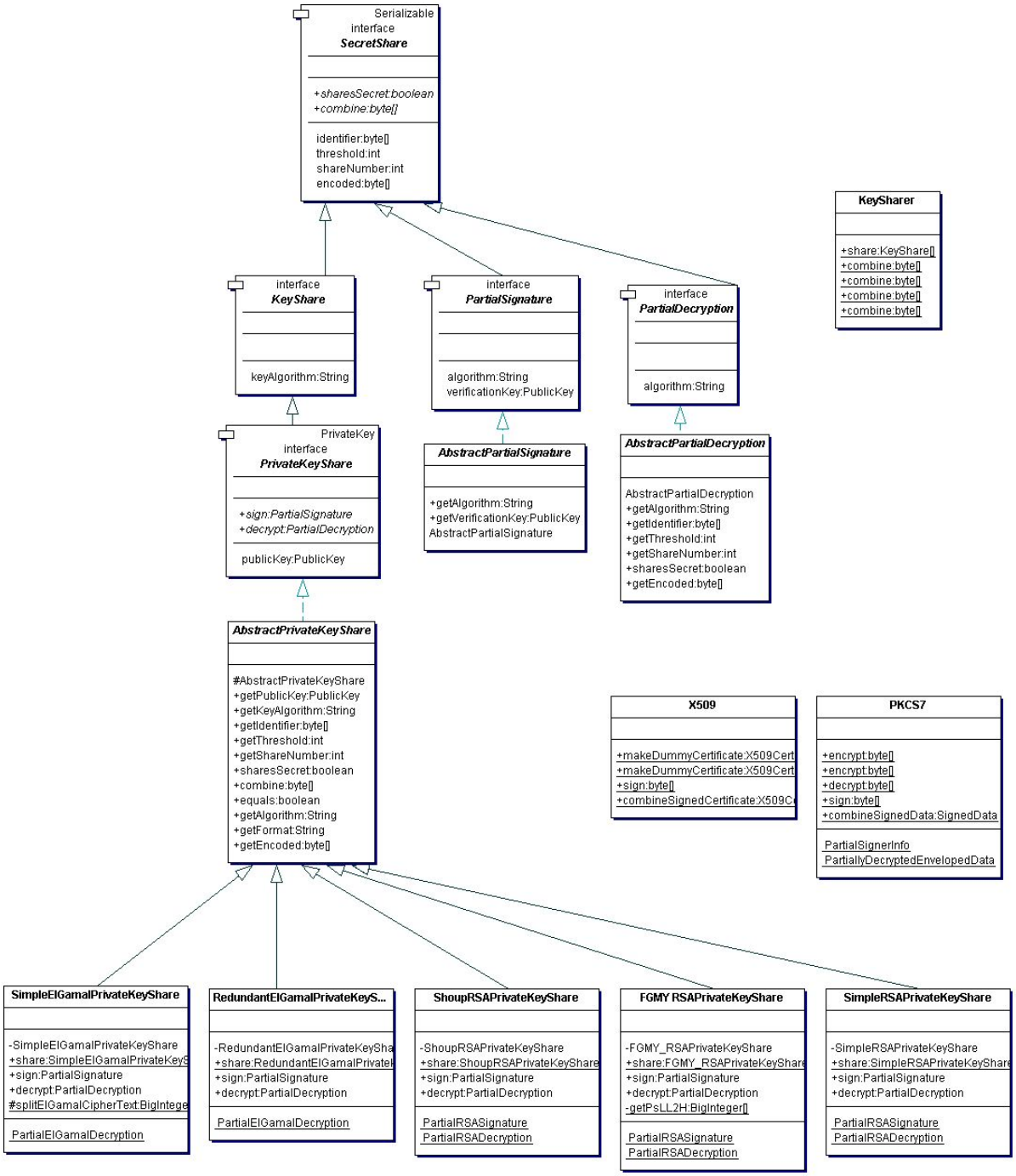


Abbildung 5.2: Key-Sharing-Basisklassen aus dem Paket `keyshare`

ist.

`PrivateKeyShare` diese Schnittstelle erweitert `KeyShare` und gleichzeitig die JCA-Schnittstelle `PrivateKey` um Methoden, mit denen der zugehörige öffentliche Schlüssel eines Schlüsselanteils erfragt werden sowie Teilsignaturen und Teilentschlüsselungen für ein zu übergebendes Byte-Array erzeugt werden können.

`AbstractPrivateKeyShare` diese abstrakte Klasse stellt Default-Implementierungen für die Methoden eines `PrivateKeyShare` bereit und dient damit zur Vermeidung von Code-Duplikaten. Insbesondere implementiert sie eine Kodierung von Schlüsselanteilen im PKCS8-Austauschformat.

`PartialSignature` diese Schnittstelle erweitert `SecretShare` und repräsentiert das Teilergebnis einer verteilten Signatur. Es sind Methoden vorgesehen um den Verifikationsschlüssel zu erhalten sowie den Namen des Signaturalgorithmus.

`AbstractPartialSignature` diese abstrakte Klasse stellt Default-Implementierungen für die Methoden einer `PartialSignature` bereit und dient damit zur Vermeidung von Code-Duplikaten. Sie erbt dabei von `AbstractSecretShare`.

`PartialDecryption` diese Schnittstelle erweitert `SecretShare` und repräsentiert das Teilergebnis einer verteilten Entschlüsselung. Es ist eine Methode vorgesehen um den Namen des Verschlüsselungsalgorithmus zu erfragen.

`AbstractPartialDecryption` diese abstrakte Klasse stellt Default-Implementierungen für die Methoden einer `PartialDecryption` bereit und dient damit zur Vermeidung von Code-Duplikaten.

`SimpleRSAPrivateKeyShare` diese Klasse stellt einen durch das einfache Verfahren zum Teilen von RSA-Schlüsseln (Abschnitt 3.4) entstandenen Teilschlüssel dar. Sie enthält auch Methoden zum Verteilen eines `RSAPrivateCrtKey` nach diesem Verfahren, sowie innere Klassen für die zugehörigen `PartialSignature` und `PartialDecryption`.

`SimpleElGamalPrivateKeyShare` diese Klasse stellt einen durch das einfache Verfahren zum Teilen von ElGamal-Schlüsseln (Abschnitt 3.8) entstandenen Teilschlüssel dar. Sie enthält auch Methoden zum Verteilen eines `ElGamalPrivateKey` nach diesem Verfahren, sowie eine innere Klasse für die zugehörige `PartialDecryption`.

`FGMY_RSAPrivateKeyShare` diese Klasse stellt einen durch das Verfahren zum Teilen von RSA-Schlüsseln nach Frankel, Gemmell, MacKenzie und Yung (Abschnitt 3.5.2) entstandenen Teilschlüssel dar. Sie enthält auch Methoden zum Verteilen eines `RSAPrivateCrtKey` nach diesem Verfahren, sowie innere Klassen für die zugehörigen `PartialSignature` und `PartialDecryption`.

`ShoupRSAPrivateKeyShare` diese Klasse stellt einen durch das Verfahren zum Teilen von RSA-Schlüsseln nach Shoup (Abschnitt 3.6) entstandenen Teilschlüssel dar. Sie enthält auch Methoden zum Verteilen eines `StrongRSAPrivateCrtKey` nach diesem Verfahren, sowie innere Klassen für die zugehörigen `PartialSignature` und `PartialDecryption`.

`RedundantElGamalPrivateKeyShare` diese Klasse stellt einen durch das redundante Verfahren zum Teilen von speziellen ElGamal-Schlüsseln (Abschnitt 3.8) entstandenen Teilschlüssel dar. Sie enthält auch Methoden zum Verteilen eines `ExtendedElGamalPrivateKey` nach diesem Verfahren, sowie eine innere Klasse für die zugehörige `PartialDecryption`.

`KeySharer` diese Klasse enthält statische Methoden zum einfachen Zugriff auf die Key-Sharing-Funktionalität (Erzeugen von Teilschlüsseln, Kombinieren von Teilsignaturen und -entschlüsselungen). Sie bietet damit eine Fassade für den Key-Sharing-Teil des Pakets.

`X509` diese Klasse enthält statische Hilfsmethoden im Zusammenhang mit X.509-Zertifikaten. Sie kapselt damit Aufrufe an das ASN1-Codec.

`PKCS7` diese Klasse enthält statische Hilfsmethoden im Zusammenhang mit PKCS7-Dateien. Dies ist ein standardisiertes Format für verschlüsselte und/oder signierte Daten. Die Klasse kapselt Aufrufe an das ASN1-Codec.

JCA-kompatible Schlüsselkodierung Zur Integration in die JCA war es von entscheidender Bedeutung, daß die privaten Schlüsselanteile von Komponenten wie `KeyStore` und `Signature` verwendet werden können. Hierzu waren zwei Maßnahmen notwendig: die Unterstützung der JCA-Schnittstelle `PrivateKey` und eine Kodierung gemäß PKCS8. Dies erlaubt es nämlich beispielsweise einem `KeyStore` den Schlüssel zu serialisieren und wieder zu rekonstruieren. PKCS8 identifiziert Schlüsselalgorithmen anhand eines Object Identifiers (OID). Der `KeyStore` sucht beim Laden von Schlüsseln dann über JCA eine `KeyFactory` für diesen OID, die wir folglich ebenfalls implementierten. Der verwendete OID ist 1.3.6.1.4.1.8301.3.2.99 und entstammt einem der TU Darmstadt zugeordneten Nummernkreis. Die PKCS8-Kodierung selbst enthält diesen OID und den Teilschlüssel als serialisiertes Java-Objekt.

5.2.2 Paket `keyshare.jca`

Das Paket `keyshare.jca` enthält Hüllen- und Adapterklassen mit denen eine möglichst große Kompatibilität zur Java Cryptographic Architecture hergestellt werden soll. Das Paket enthält Implementierungen von `Cipher`, `Signature` und `KeyFactory` zum Umgang mit Schlüsselanteilen sowie einen JCA-Provider, der die neuen Algorithmen anmeldet und dadurch verfügbar macht. Darüberhinaus werden die speziellen Schlüssel, die vom Shoup- und vom redundanten ElGamal-Key-Sharing-Verfahren benötigt werden, bereitgestellt. Die Struktur der im folgenden kurz beschriebenen Klassen und Schnittstellen ist in Abbildung 5.3 dargestellt.

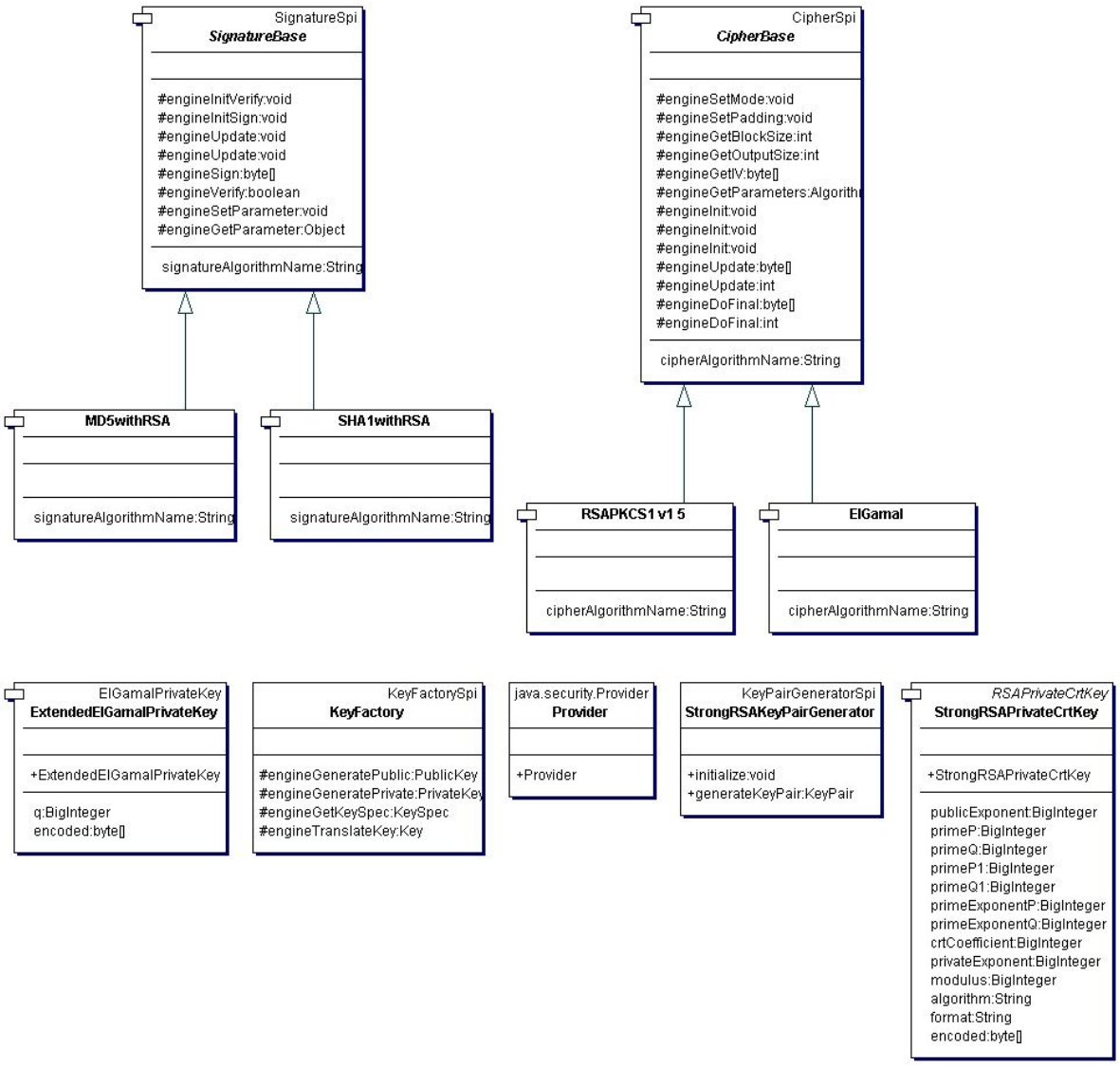


Abbildung 5.3: JCA-Kompatibilitätsklassen aus dem Paket `keyshare.jca`

`Provider` dieser JCA-Provider meldet die implementierten Algorithmen entsprechend der JCA-Namenskonventionen an.

`StrongRSAPrivateCrtKey` diese Klasse erweitert einen JCA-`RSAPrivateCrtKey` und repräsentiert einen privaten RSA-Schlüssel mit zwei Primfaktoren der Gestalt $p = 2p' + 1$ und $q = 2q' + 1$. Solche Schlüssel werden vom Shoup-Verfahren benötigt.

`ExtendedElGamalPrivateKey` diese Klasse erweitert `ElGamalPrivateKey` aus dem FlexiProvider und repräsentiert einen privaten ElGamal-Schlüssel mit einem Modul der Form $p = mq+1$, wie er vom redundanten ElGamal-Key-Sharing-Verfahren benötigt wird.

`KeyFactory` diese Klasse dient zum Erzeugen von `PrivateKeyShare`-Instanzen aus ihren PKCS8-Repräsentationen.

`StrongRSAKeyPairGenerator` dieser Schlüsselgenerator erzeugt RSA-Schlüsselpaare mit einem `StrongRSAPrivateCrtKey`.

`SignatureBase` diese abstrakte Klasse stellt Default-Implementierungen für die Methoden eines JCA-`SignatureSpi` bereit und dient damit zur Vermeidung von Code-Duplikaten. Es ist vorgesehen, zur Verifikation eine `Signature`-Instanz eines anderen Providers zu kapseln und die Erzeugung einer Teilsignatur an den jeweiligen `PrivateKeyShare` zu delegieren.

`MD5withRSA` diese konkrete Implementierung von `SignatureBase` bietet verteilte RSA-Signaturen mit MD5-Hashing. Die Hashfunktion und die Verifikation werden dabei an andere Provider delegiert, müssen also bereits vorhanden sein.

`SHA1withRSA` diese konkrete Implementierung von `SignatureBase` bietet verteilte RSA-Signaturen mit SHA1-Hashing. Die Hashfunktion und die Verifikation werden dabei an andere Provider delegiert, müssen also bereits vorhanden sein.

`CipherBase` diese abstrakte Klasse stellt Default-Implementierungen für die Methoden eines JCE-`CipherSpi` bereit und dient damit zur Vermeidung von Code-Duplikaten. Es ist vorgesehen, zur Verschlüsselung eine `Cipher`-Instanz eines anderen Providers zu kapseln und die Erzeugung einer Teilentschlüsselung an den jeweiligen `PrivateKeyShare` zu delegieren.

`RSAPKCS1_v1_5` diese konkrete Implementierung von `CipherBase` erzeugt verteilte RSA-Entschlüsselungen. Es wird dabei das PKCS1-Padding in der Version 1.5 verwendet. Die Verschlüsselung wird dabei an andere Provider delegiert, muß also bereits vorhanden sein.

`ElGamal` diese konkrete Implementierung von `CipherBase` erzeugt verteilte ElGamal-Entschlüsselungen. Die Verschlüsselung wird dabei an andere Provider delegiert, muß also bereits vorhanden sein.

5.2.3 Paket `keyshare.keystore`

Das Paket `keyshare.keystore` enthält zwei `KeyStore`-Implementierungen, die von Secret- und Key-Sharing Gebrauch machen. Der ShamirStore kapselt einen anderen Software-Keystore (beispielsweise den JKS von Sun) und verteilt ihn zur Speicherung in mehrere Dateien nach dem Shamir-Verfahren. Der KeySharer verteilt die einzelnen Schlüssel, die in ihn gesteckt werden, mit jeweils dafür geeigneten Key-Sharing-Verfahren auf mehrere abhängige Keystores. Diese Keystores können dann zum Erstellen von Teilsignaturen und -entschlüsselungen genutzt werden. Der Shamir-Store verfügt über eine kleine graphische Benutzerschnittstelle, die sich im Unterpaket `keyshare.keystore.gui` befindet. Beide Keystores werden über XML-Dateien konfiguriert, deren Zugriffspfade im Unterpaket `keyshare.keystore.xml` definiert sind.

`KeyStoreBase` diese abstrakte Klasse stellt Default-Implementierungen für die Methoden eines JCA-`KeyStoreSpi` bereit und dient damit zur Vermeidung von Code-Duplikaten. Sie bedient sich dabei einer gekapselten `KeyStore`-Instanz aus einem anderen Provider.

`ShamirStore` diese von `KeyStoreBase` abgeleitete Klasse implementiert den ShamirStore-Keystore.

`KeySharer` diese von `KeyStoreBase` abgeleitete Klasse implementiert den KeySharer-Keystore.

Unterpaket `keyshare.keystore.xml` Die Keystores werden über XML-Dateien konfiguriert, in denen beispielsweise angegeben wird auf wieviele und welche Dateien der Keystore seinen Inhalt verteilen soll (eine Beschreibung der möglichen Einstellungen findet sich in Abschnitt 5.4). Diese Dateien werden mit Hilfe des Castor-XML-Frameworks ausgewertet. Dieses stellt ein Mapping zwischen XML-Tags und Java-Klassen bereit. Die entsprechenden Klassen liegen im Paket `keyshare.keystore.xml`:

`ShamirStore` diese Klasse entspricht dem `<shamir-store>`-XML-Tag, welches das Wurzel-Element für die Konfigurationsdatei des ShamirStore darstellt. Es besitzt einige Attribute und kann geschachtelte `<share>`-Tags aufnehmen.

`KeySharer` diese Klasse entspricht dem `<key-sharer>`-XML-Tag, welches das Wurzel-Element für die Konfigurationsdatei des KeySharers darstellt. Es besitzt einige Attribute und kann geschachtelte `<share>`-Tags aufnehmen.

`Share` diese Klasse entspricht dem `<share>`-Tag, welches in beiden Konfigurationsdateien verwendet wird, um den Anteilen ihre Dateien zuzuordnen.

Konfiguration nur über Dateien Die beiden KeyStores können nur über die XML-Dateien konfiguriert werden. Dies geschieht durch Einlesen der Dateien während der Methode `KeyStore.load(InputStream, char[])`. Für eine

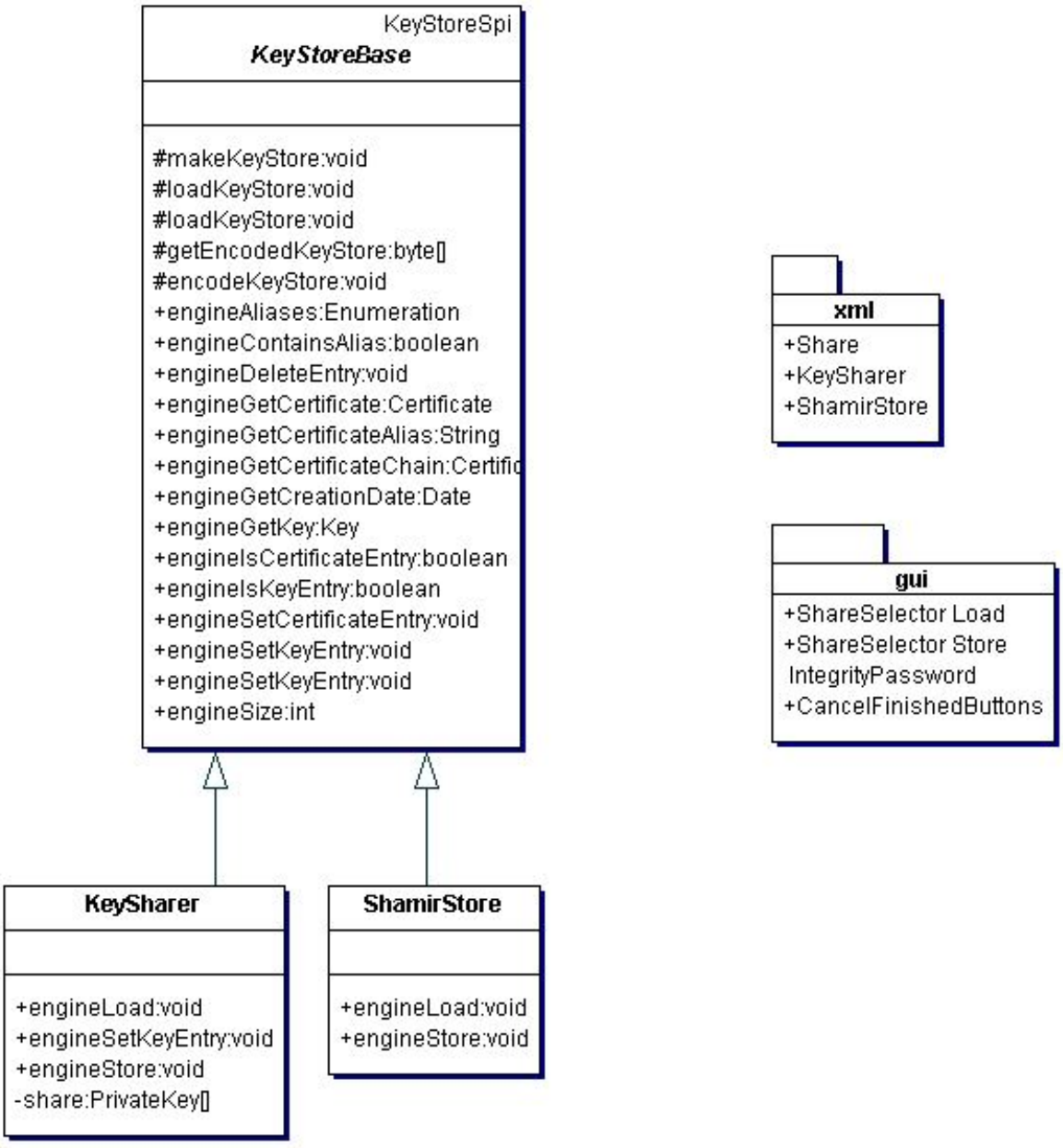


Abbildung 5.4: Klassen aus dem Paket `keyshare.keystore`

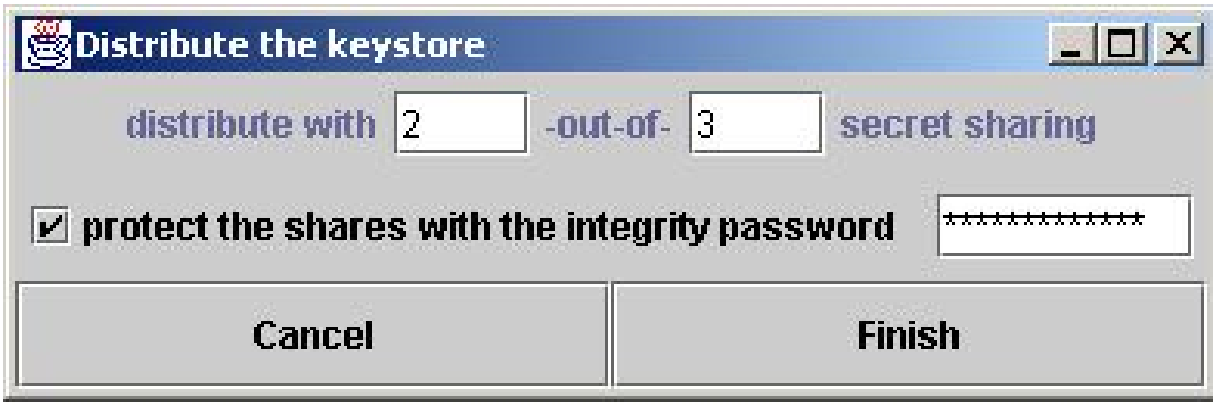


Abbildung 5.5: Dialog zum Anlegen von Anteilen des Shamir-Store

direkte Kontrolle des KeyStores durch das Programm, in dem es verwendet wird, ist dies etwas umständlich. In einem solchen Fall muß zunächst eine entsprechende XML-Datei (zumindest im Speicher) erzeugt werden. Angenehmer wären Methodenaufrufe auf der KeyStore-Instanz zur Parameterwahl, etwa eine Methode `setThreshold(int)`. Dies ist allerdings aufgrund der Vorgaben durch die JCA nicht möglich: Unsere Implementierung wird als `KeyStoreSpi` von JCA-internen Klassen instantiiert und nicht für die Anwendung zugänglich gemacht. Eine Erweiterung der Schnittstelle über die seitens JCA vorgegebene hinaus ist damit nicht möglich.

Unterpaket `keyshare.keystore.gui` Der ShamirStore verfügt über eine graphische Benutzeroberfläche mit dessen Hilfe er den Benutzer fragen kann, in welche Dateien die Anteile geschrieben beziehungsweise aus welchen Dateien sie gelesen werden sollen. Diese Oberfläche wurde mit dem Java-eigenen Swing-Framework implementiert und befindet sich im Paket `keyshare.keystore.gui`.

`ShareSelector_Load` diese von `JFrame` abgeleitete Klasse implementiert ein Fenster in dem der Benutzer die vorgegebene Zahl an Dateien zur Rekonstruktion des ShamirStore auswählen und (anhand des Integritätspaßworts) überprüfen kann.

`ShareSelector_Store` diese von `JFrame` abgeleitete Klasse bringt ein Fenster auf den Bildschirm in dem der Benutzer die Anzahl der Anteile und das Integritätspaßwort einstellen kann und anschließend die Zieldateien auswählt.

`IntegrityPassword` diese von `JPanel` abgeleitete Klasse stellt eine Eingabefläche für das Integritätspaßwort dar. Sie wird von den beiden Selector-Klassen verwendet.

`CancelFinishedButton` dieses `JPanel` beinhaltet eine Eingabefläche mit den beiden Schaltflächen zum Abbrechen und Bestätigen. Sie wird von den beiden Selector-Klassen verwendet.

Abbildung 5.6: Dialog zum Einlesen von Anteilen des Shamir-Store

5.2.4 Paket `keyshare.escrow`

Das Paket `keyshare.escrow` beschäftigt sich mit Schlüsselverwahrung und -wiederherstellung. Es enthält eine Implementierung für eine Verwahrung in Dateien, sowie spezielle `KeyPairGenerator` für wiederherstellbare RSA- und ElGamal-Schlüssel (Abschnitt 4.3).

`AbstractEscrowedKeyPairGenerator` diese abstrakte Basisklasse stellt die allgemeingültigen Methoden für einen Schlüsselgenerator für wiederherstellbare Schlüssel zur Verfügung. Dies umfaßt vor allem die Kapselung eines normalen `KeyPairGenerators` für den entsprechenden Algorithmus und die Verwaltung der zum Einsatz kommenden `Cipher`-Instanz. Die konkrete Schlüsselerzeugung wird mittels einer Schablonenmethode an die konkreten Unterklassen delegiert.

`EscrowedKeyParameterSpec` mit dieser Klasse kann der öffentliche Schlüssel, der zum Key-Escrow verwendet werden soll, übermittelt werden. Wird der `KeyPairGenerator` nicht mit einer solchen Instanz initialisiert, erzeugt er normale (nicht wiederherstellbare) Schlüsselpaare.

`ElGamalKeyPairGenerator` diese Klasse generiert wiederherstellbare ElGamal-Schlüssel.

`RSAKeyPairGenerator` diese Klasse generiert wiederherstellbare RSA-Schlüssel.

`KeyEscrowBase` diese abstrakte Basisklasse leistet die Ver- und Entschlüsselungsoperationen für die Schlüsselhinterlegung. Die Art der Speicherung (zum Beispiel in Dateien oder einer Datenbank) wird den Unterklassen überlassen.

`FileKeyEscrow` diese Klasse implementiert die Speicherung der hinterlegten Schlüssel in das Dateisystem.

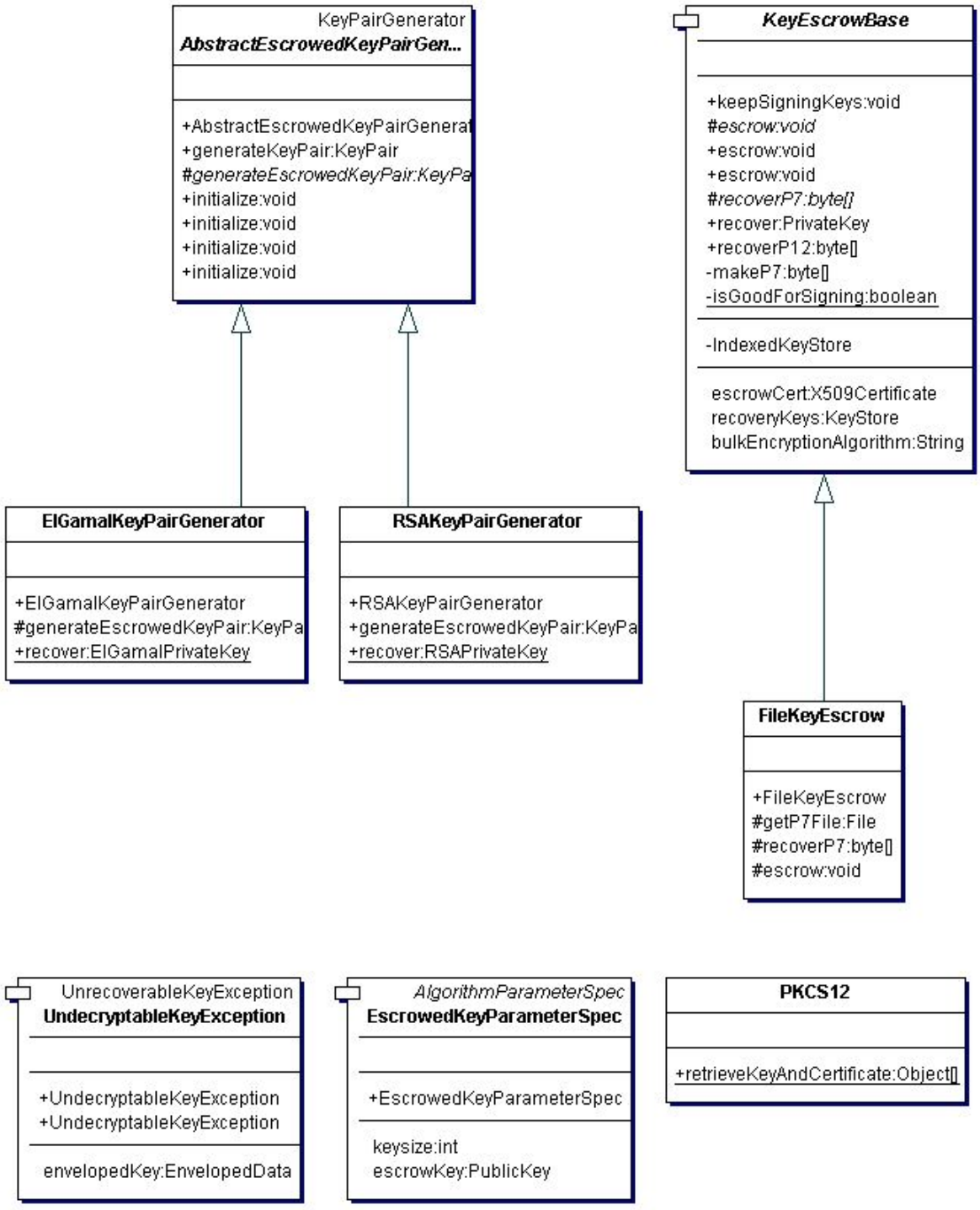


Abbildung 5.7: Klassen aus dem Paket `keyshare.escrow`

`UndecryptableKeyException` diese Ausnahme wird von den Key-Escrow-Klassen geworfen, wenn bei der Schlüsselwiederherstellung der Schlüssel nicht entschlüsselt werden konnte, zum Beispiel weil der Recovery-Schlüssel nicht vorlag. Die Klasse kapselt dabei die verschlüsselte Datei, die dann an andere Applikationen weitergereicht werden kann.

`PKCS12` diese Klasse enthält statische Methoden zum Zugriff auf PKCS12-Daten. PKCS12 ist ein Standard für Softtoken, also dateibasierte Träger vertraulicher Informationen. Die Key-Escrow-Applikation und die FlexiTrust-Trustcentersoftware benutzen PKCS12 als Transportformat für Schlüssel.

5.2.5 Paket `keyshare.apps`

Das Paket `keyshare.apps` enthält drei ausführbare Java-Kommandozeilen-Applikationen mit denen Secret- und Key-Sharing verwendet sowie Key-Escrow durchgeführt werden können. Dies sind der FileSharer, mit dem beliebige Dateien nach dem Shamir-Verfahren in mehrere Anteile aufgeteilt werden können, der KeySharer, der sich zum Erzeugen verteilter Schlüssel, von Teilsignaturen und -entschlüsselungen sowie deren Kombination eignet und KeyEscrow, mit dessen Hilfe Schlüssel hinterlegt und rekonstruiert werden können. Die Funktionsweise der Programme ist in den Abschnitten 5.4.1, 5.4.2 und 5.4.3 beschrieben.

`CommandLineApp` diese abstrakte Basisklasse stellt einige Grundfunktionalität zum Auswerten der Kommandozeilenparameter fest und bestimmt mit Schablonenmethoden den groben Ablauf der drei davon abgeleiteten Applikationen.

`ApplicationMode CommandLineApp` geht davon aus, daß eine Anwendung aus mehreren Modi besteht, von denen jeweils einer ausgewählt und durchgeführt wird. Diese Schnittstelle beschreibt die Methoden, die dabei automatisch von `CommandLineApp` aufgerufen werden.

`FileSharer` diese Klasse implementiert die Applikation FileSharer (Abschnitt 5.4.1.

`KeySharer` diese Klasse implementiert die Applikation KeySharer (Abschnitt 5.4.2.

`KeyEscrow` diese Klasse implementiert die Applikation KeyEscrow (Abschnitt 5.4.3.

5.2.6 Paket `keyshare.tests`

Das Paket `keyshare.tests` enthält Testfälle anhand derer die korrekte Funktionsweise der Klassen in den anderen Paketen überprüft werden kann. Die Testfälle sind gemäß dem Framework JUnit erstellt worden, welches sich mit Komponententests für Java-Klassen beschäftigt. Dementsprechend können sie mit entsprechenden Tools automatisch durchgeführt und ausgewertet werden. Die Pflege der Testfälle hat sich während der Entwicklung als extrem wertvoll

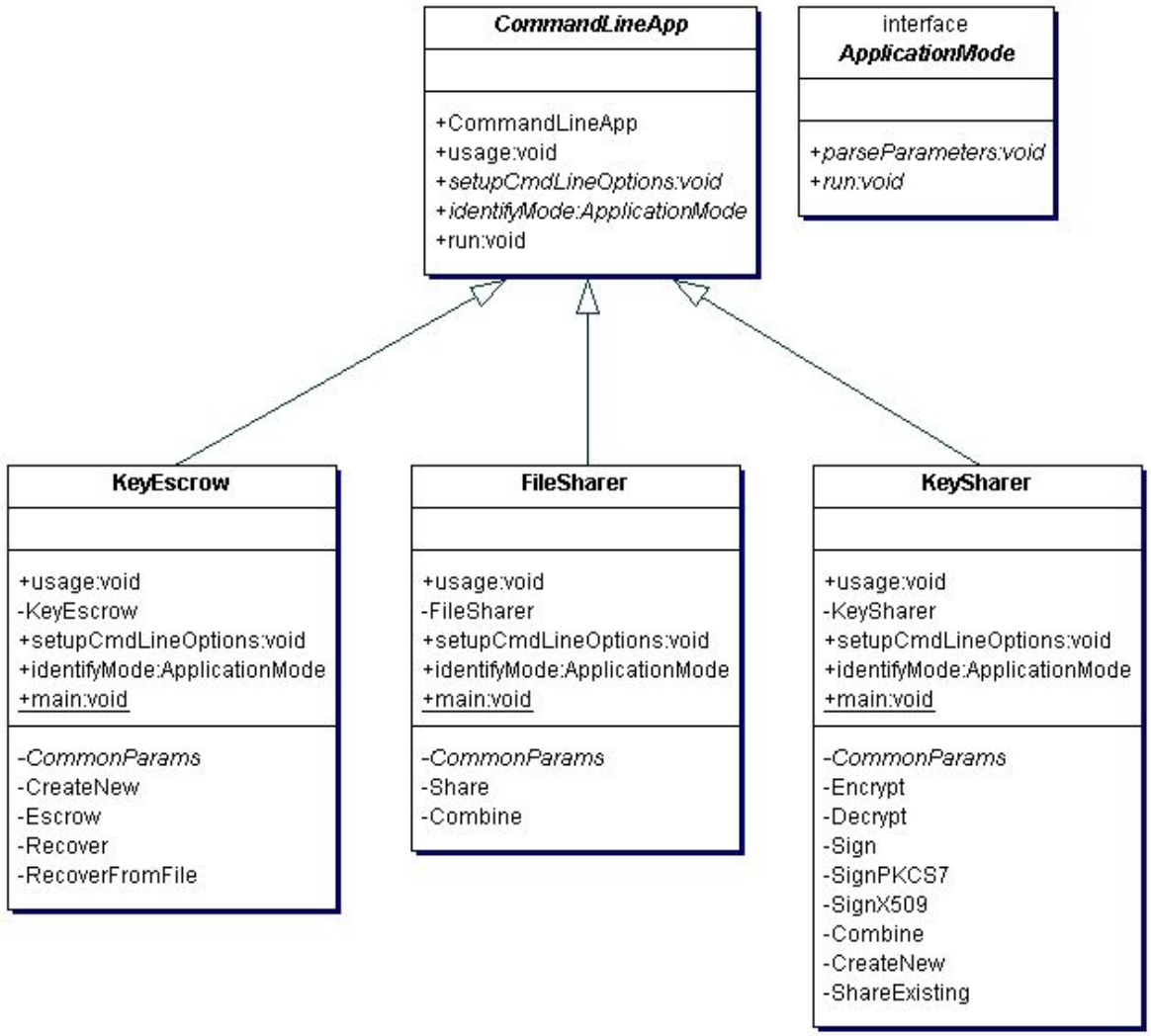


Abbildung 5.8: Klassen aus dem Paket `keyshare.apps`

bei der Fehlervermeidung, -suche und -behebung erwiesen. Insgesamt enthält das Paket 22 Testfälle.

`SecretSharerTests` diese Klasse umfaßt drei Tests für die Klasse `SecretSharer` und testet somit die Secret-Sharing-Grundfunktionalität.

`KeySharerTests` diese Klasse testet die Key-Sharing-Grundfunktionalität anhand von sechs Tests für die verschiedenen damit befaßten Klassen (vor allem `KeySharer` und die `Signature`- und `Cipher`-Implementierungen).

`ShamirStoreTests` diese Klasse testet den ShamirStore.

`KeyEscrowTests` diese Klasse testet mit zwei Tests die Key-Escrow-Funktionen.

`PKCS7Tests` diese Klasse testet Ver- und Entschlüsselung mit PKCS7.

`X509Tests` diese Klasse testet die verteilte Erzeugung und Zusammenführung von X.509-Zertifikaten.

`AppTests` diese Klasse testet mit acht Tests die Applikationen FileSharer, KeySharer und KeyEscrow.

5.2.7 Benötigte Klassenbibliotheken

Unsere Implementierung greift auf eine Reihe von Java-Klassenbibliotheken zurück, die Basisfunktionalität aus verschiedenen Bereichen bereitstellen. Alle diese Bibliotheken müssen in Form von JAR-Dateien vorliegen.

Kryptographie-Erweiterungen falls eine Java-Umgebung vor Version 1.4 verwendet wird, muß die JCE nachinstalliert werden.

FlexiCore-Provider zur Verwendung von ElGamal-Schlüsseln muß der FlexiCore-Provider installiert sein. Dieser wird vom Fachgebiet Theoretische Informatik entwickelt und ist auch doch frei erhältlich. Auch sonst empfiehlt sich dessen Verwendung, da sich während der Entwicklung in einigen Bereichen Probleme mit dem Provider von Sun ergaben.

ASN.1-Codec zum Umgang mit den ASN.1-Datenstrukturen wird das Codec-Paket des Fraunhofer-Instituts verwendet, das mittlerweile ebenfalls vom Fachgebiet gepflegt wird.

XML die Verarbeitung von XML-Dateien wird Castor überlassen, einem Open-Source-Paket zum Transformieren von Java-Klassen in (unter anderem) XML.

Kommandozeilen-Parser die Kommandozeilen-Applikationen verwenden die Bibliothek jargs zum Auswerten der übergebenen Parameter

JUnit zum Ausführen der Testklassen wird das Testframework JUnit benötigt.

5.3 Programmierschnittstelle

5.3.1 Secret-Sharing-Basisfunktionalität

Die grundlegenden Secret-Sharing-Funktionen werden durch die statischen Methoden der Klasse `keyshare.SecretSharer` zugänglich gemacht.

Geheimnis verteilen

```
public static SecretShare[] share
(byte[] secret, int threshold, int sharenumber)
throws InvalidParameterException, NoSuchAlgorithmException
```

Diese Methode erzeugt für das Bytefeld `secret` insgesamt `sharenumber` Anteile, von denen `threshold` zur Rekonstruktion benötigt werden. Ob dabei die XOR-Methode oder das Shamir-Verfahren zum Einsatz kommt wird automatisch anhand des `threshold` entschieden Ungültige Werte für `sharenumber` und `threshold` führen zu einer `InvalidParameterException`, falls auf dem System keine MD5-Hashfunktion (zur Berechnung der ID für die Anteile) verfügbar ist, wird eine `NoSuchAlgorithmException` ausgelöst.

Integritätspaßwort setzen

```
public static void protectIntegrity
(PasswordIntegrityProtected[] shares, String password)
throws NoSuchAlgorithmException
```

Bevor der Geber die Anteile an die Teilnehmer weiterreicht, kann er sie mit einem Integritätspaßwort versehen. Um das von der Methode `share(byte[], int, int)` erhaltene `SecretShare[]` übergeben zu können, ist ein Typecast notwendig. Bei den von der genannten Methode erzeugten `SecretShare[]` ist dies stets möglich. Falls die MD5-Funktion, auf der der Integritätsschutz beruht nicht verfügbar ist, wird eine `NoSuchAlgorithmException` ausgelöst.

Anteile serialisieren

```
byte[] bytes = share.getEncoded();
```

Jeder `SecretShare` verfügt über eine Methode `byte[] getEncoded()`, die ihn in eine serialisierte Form überführt, in der er leicht transportiert werden kann. Da hierbei Java-Objektserialisierung zum Einsatz kommt, kann es mit einem `java.io.ObjectInputStream` aus dieser Form wieder instantiiert werden.

Anteile zusammensetzen

```
public static byte[] combine
(SecretShare[] shares)
throws NoSuchAlgorithmException, SecretSharingException

public static byte[] checkAndCombine
```

```
(SecretShare[] shares, String integrityPassword)
throws NoSuchAlgorithmException, SecretSharingException
```

Diese beiden Methoden kombinieren die ihnen übergebenen Anteile zu dem Bytefeld, aus dem diese hervorgegangen sind. `checkAndCombine` überprüft dabei zuvor die Integrität der Anteile anhand eines Paßwortes. Wenn das Geheimnis aufgrund fehlerhafter Eingaben nicht rekonstruiert werden konnte, wird eine `SecretSharingException` ausgelöst, wenn die MD5-Funktion, die benötigt wird, um die Korrektheit des Geheimnisses zu prüfen, fehlt, gibt es eine `NoSuchAlgorithmException`.

5.3.2 Key-Sharing-Basisfunktionalität

Die grundlegenden Key-Sharing-Funktionen werden durch die statischen Methoden der Klasse `keyshare.KeySharer` zugänglich gemacht.

Schlüssel verteilen

```
public static KeyShare[] share
(Key secret, int threshold, int sharenumber)
throws InvalidParameterException, NoSuchAlgorithmException,
    InvalidKeySpecException
```

Diese Methode erzeugt für den übergebenen Schlüssel `secret` insgesamt `sharenumber` Anteile, von denen `threshold` zur Rekonstruktion benötigt werden. Das dabei eingesetzte Verfahren hängt von der Art des Schlüssels sowie von den Werten für `sharenumber` und `threshold` ab. Ungültige Werte für `sharenumber` und `threshold` führen zu einer `InvalidParameterException`, nicht unterstützte Schlüsseltypen verursachen eine `NoSuchAlgorithmException` oder eine `InvalidKeySpecException`.

Teilsignaturen und Teilentschlüsselungen erstellen

```
Signature signer = Signature.getInstance("DistributedMD5withRSA");
signer.initSign(keyshare);
signer.update(message);
byte[] partialSignature = signer.sign();
```

Da die erzeugten Instanzen von `KeyShare` für alle unterstützten Arten von `java.security.PrivateKey` selbst die Schnittstelle `PrivateKey` implementieren, können die Teilberechnungen JCA-konform über `java.security.Signature` und `javax.crypto.Cipher` erstellt werden. Hierzu müssen die entsprechenden Implementierungen des `keyshare.jca.Provider` verwendet werden, der folglich im Laufzeitsystem registriert werden sollte. Implementiert sind die Signaturen `DistributedMD5withRSA`, `DistributedSHA1withRSA` sowie die Chiffren `DistributedRSA` und `DistributedElGamal`.

Teilberechnungen zusammenführen

```
public static byte[] combine
(byte[][] shares, byte[] message)
throws NoSuchAlgorithmException, SecretSharingException
```

Die Teiberechnungen können direkt als zwei-dimensionales Bytefeld übergeben werden. Der zweite Parameter `message` kann bei Signaturen verwendet werden, um die entstandene Signatur automatisch zu verifizieren. Bei Entschlüsselungen wird er nicht ausgewertet. Wenn die Kombination aufgrund fehlerhafter Eingaben nicht rekonstruiert werden konnte, wird eine `SecretSharingException` ausgelöst, wenn die Verifikations-Funktion für Signaturen fehlt, gibt es eine `NoSuchAlgorithmException`.

5.3.3 Der Shamir-KeyStore

Der `ShamirStore` ist ein spezieller `KeyStore`, der seinen Inhalt automatisch auf mehrere Dateien verteilt, aus denen er auch wieder geladen werden kann. Dieser Vorgang ist aufgrund der Einbettung in das `KeyStore`-Interface der JCA für den Anwender transparent. Konfiguriert wird der `ShamirStore` mit einer XML-Datei.

Konfiguration Der `ShamirStore` bezieht seine Parametrisierung aus einer XML-Datei, die ihm während dem Ladevorgang über einen Eingabestrom zugeführt werden muß. Dieser etwas umständliche Weg ist durch die seitens JCA vorgegebene Schnittstelle bedingt. Die XML-Datei hat folgenden Aufbau:

```
<shamir-store threshold="2" load="AUTO store="AUTO"
        integrity-password="Passwort">
        <share file="1a.share" />
        <share file="2b.share" />
        <share file="3c.share" />
</shamir-store>
```

Für jeden Anteil, der erzeugt werden soll, gibt es ein geschachteltes `<share />`-Tag, welches den Namen der Datei angibt, die diesem Anteil zugeordnet wird. Die Anzahl der Anteile die benötigt werden, um den `ShamirStore` zu rekonstruieren, wird durch das Attribut `threshold` festgelegt. Das optionale Attribut `integrity-password` gibt das Integritätspaßwort zum Schutz der Anteile an. Die beiden Attribute `load` und `store` bestimmen das Verhalten des `ShamirStore` beim Laden und Zurückschreiben. Mit der Einstellung `AUTO` werden die entsprechenden Dateien automatisch ausgelesen beziehungsweise angelegt. Die Einstellung `OFF` führt dazu, daß der `ShamirStore` nicht geladen oder gespeichert wird. Sie kann sinnvoll sein, wenn ein versehentliches Überschreiben der Anteile durch eine Art Nur-Lesen-Modus verhindert werden soll, oder wenn die Anteile neu erzeugt werden sollen ohne zuvor die (eventuell nicht existenten) Dateien einzulesen. Die Einstellung `INTERACTIVE` aktiviert die graphische Benutzeroberfläche bei jedem Lade- und Speichervorgang. Auf diese Weise kann der Benutzer die Dateien auswählen.

Instanz erzeugen

```
KeyStore ks = KeyStore.getInstance("ShamirStore");
```

Keystore laden Der `ShamirStore` setzt sich aus den in der Konfigurationsdatei bezeichneten Anteilen zusammen, wenn seine `load()`-Methode aufgerufen wird. Die Konfigurationsdatei muß ihm über den `InputStream` dieser Methode zugeführt werden. Ein ebenfalls angegebenes Paßwort wird dazu verwendet, die Integrität der KeyStore-Datei, die sich aus den Anteilen ergibt, zu prüfen. Dies ist zu unterscheiden von dem Integritätspaßworts für die einzelnen Anteile, welches in der XML-Datei angegeben werden kann.

```
FileInputStream in = new FileInputStream("configuration.xml");
ks.load(in, password);
in.close();
```

Wenn das Laden aufgrund fehlender oder fehlerhafter Anteile fehlschlägt wird eine `IOException` geworfen.

Keystore verwenden Der `ShamirStore` verwendet intern einen normalen JavaKeyStore, an den er alle Anfragen weiterreicht. Daher reagiert er genau so, wie man es von einem solchen erwarten würde. Unter anderem kann man einen Schlüssel mit einem Alias versehen paßwortgeschützt abspeichern und auch wieder laden:

```
ks.setKeyEntry(alias, key, password, certificateChain);
Key key = ks.getKey(alias, password);
Certificate cert = ks.getCertificate(alias);
```

Keystore speichern Der `ShamirStore` speichert den intern verwendeten JavaKeyStore zunächst in ein Bytefeld ab und verteilt dieses dann über den `SecretSharer` auf die angegeben Dateien. Der Ausgabestrom, der der `store()`-Methode übergeben wird, wird dabei ignoriert. Das Paßwort allerdings wird zum Integritätsschutz des JavaKeyStores verwendet, wenn dieser in das Bytefeld geschrieben wird.

```
ks.store(null,  password);
```

5.3.4 Der KeySharer-KeyStore

Der `KeySharer` ist ein spezieller Keystore, der die ihm anvertrauten Schlüssel mit einem Key-Sharing-Verfahren verteilt und in mehrere abhängige Keystores speichert. Die entstehenden Keystores können unabhängig voneinander verwendet werden, um mit den Teilschlüsseln zu arbeiten. Im Gegensatz zu den dabei entstehenden Teilergebnissen ist es nicht vorgesehen, die Schlüssel selbst wieder zusammenzusetzen. Dementsprechend sollten die Methoden, die einen Schlüssel aus dem Keystore laden für den `KeySharer` nicht aufgerufen werden. Statt dessen müssen die abhängigen Keystores direkt und einzeln verwendet werden.

Konfiguration Der `KeySharer` bezieht seine Parametrisierung aus einer XML-Datei, die ihm während dem Ladevorgang über einen Eingabestrom zugeführt werden muß. Die XML-Datei hat folgenden Aufbau:

```
<key-sharer threshold="2" >
        <share file="1a.keystore" />
        <share file="2b.keystore" />
        <share file="3c.keystore" />
</key-sharer>
```

Für jeden Anteil, der erzeugt werden soll, gibt es ein geschachteltes `<share />`-Tag, welches den Namen der Datei angibt, die diesem Anteil zugeordnet wird. Die Anzahl der aus Anteilen erzeugten Teilsignaturen oder -entschlüsselungen die benötigt werden, um das Gesamtergebnis erhalten zu können, wird durch das Attribut `threshold` festgelegt.

Instanz erzeugen

```
KeyStore ks = KeyStore.getInstance("KeySharer");
```

Keystore laden Der `KeySharer` lädt die abhängigen Keystores aus den in der Konfigurationsdatei bezeichneten Dateien wenn seine `load()`-Methode aufgerufen wird. Die Konfigurationsdatei muß ihm über den `InputStream` dieser Methode zugeführt werden. Ein ebenfalls angegebenes Paßwort wird dazu verwendet, die Integrität dieser Keystores zu prüfen. Obwohl die abhängigen Keystores geladen werden, werden die darin enthaltenen Teilschlüssel nicht wieder zusammengesetzt. Das Laden dient nur dazu, die Keystores um neue Schlüssel zu ergänzen.

```
FileInputStream in = new FileInputStream("configuration.xml");
ks.load(in, password);
in.close();
```

Wenn das Laden aufgrund fehlender oder fehlerhafter Anteile fehlschlägt wird eine `IOException` geworfen.

Keystore verwenden Der `KeySharer` eignet sich nur zum Speichern von Schlüsseln. Zertifikatseinträge können nicht angelegt werden, außerdem können Schlüssel nicht wieder geladen werden. Die Schlüssel werden automatisch mit einem geeigneten Key-Sharing-Verfahren auf die abhängigen Keystores verteilt. Wenn es kein solches Verfahren für den vorliegenden Schlüsseltyp gibt, wird eine Ausnahme ausgelöst.

```
ks.setKeyEntry(alias, key, password, certificateChain);
```

Keystore speichern Der `KeySharer` speichert die abhängigen Keystores intern zwischen und schreibt sie erst in ihre Dateien zurück, wenn die Methode `store()` aufgerufen wird. Der Ausgabestrom, der der `store()`-Methode übergeben wird, wird dabei ignoriert. Das Paßwort allerdings wird zum Integritätsschutz der abhängigen Keystores verwendet.

```
ks.store(null,  password);
```

5.3.5 Das Key-Escrow-System

Das Key-Escrow-System besteht aus zwei unabhängigen Komponenten. Zum einen können Schlüsselpaare erzeugt werden, aus denen zum Recovery der private aus dem öffentlichen Schlüssel berechnet werden kann. Hier kommen die Verfahren aus Abschnitt 4.3 zum Einsatz. Auf der anderen Seite können private Schlüssel in verschlüsselten Dateien hinterlegt und so später wiederhergestellt werden. Der zweite Ansatz ist universell einsetzbar, während der erste nur für bestimmte Schlüssel geeignet ist.

Wiederherstellbares Schlüsselpaar erzeugen Um ein wiederherstellbares Schlüsselpaar für RSA oder ElGamal zu erzeugen, werden entsprechende `KeyPairGenerator` per JCA angefordert und konfiguriert. Bei der Initialisierung muß der öffentliche Schlüssel des Recovery-Operators angegeben werden. Mit diesem werden die Informationen über den privaten Schlüssel dann chiffriert und in den öffentlichen Schlüssel eingebracht. Es ist nicht notwendig, daß dieser Schlüssel und das erzeugte Schlüsselpaar dem gleichen Algorithmus angehören. Außerdem wird die Bitlänge des zu erzeugenden Schlüssels angegeben. Hierbei ist darauf zu achten, daß sie groß genug ist, um die chiffrierten Daten unterzubringen (also mindestens so groß wie der Escrow-Schlüssel).

```
KeyPairGenerator keygen = KeyPairGenerator.getInstance
          ("RSA", "KeySharingProvider");
keygen.initialize(new EscrowedKeyParameterSpec
          (1024, (PublicKey)escrow));
KeyPair pair = keygen.generateKeyPair();
```

privaten Schlüssel wiederherstellen Für die Schlüsselwiederherstellung sieht JCA keine Schnittstelle vor. Statt dessen müssen statische Methoden der entsprechenden `KeyPairGenerator`-Klassen aufgerufen werden. Dieses wird der zugehörige öffentliche Schlüssel des gewünschten privaten Schlüssels übergeben sowie das Escrow- und Recovery-Schlüsselpaar.

```
PrivateKey recovered = RSAKeyPairGenerator.recover(
(RSAPublicKey)key, (PrivateKey)recovery, (PublicKey)escrow);
```

privaten Schlüssel im Dateisystem verwahren Die Klasse `FileKeyEscrow` speichert die ihr übergebenen Schlüssel als chiffrierte PKCS7-Dateien in einem Verzeichnis im Dateisystem ab. Die Schlüssel werden dabei durch die Aussteller und Seriennummern der zugehörigen Zertifikate identifiziert. Sowohl das

Verzeichnis als auch der Escrow-Schlüssel, mit dem die PKCS7-Dateien verschlüsselt werden, sind frei wählbar.

```
FileKeyEscrow escrow = new FileKeyEscrow((File)escrowDir);
escrow.setEscrowCert((X509Certificate)cert);
escrow.escrow((Key)key, (X509Certificate)userCert);
```

Normalerweise speichert der `FileKeyEscrow` keine Schlüssel, die zur Ausstellung digitaler Signaturen geeignet sind (dies ist anhand des Zertifikats ersichtlich). Statt dessen erzeugt er eine Ausnahme. Sollen dennoch Signaturschlüssel verwahrt werden, kann dieses Verhalten explizit unterdrückt werden.

privaten Schlüssel aus dem Dateisystem wiederherstellen Die Wiederherstellung der privaten Schlüssel aus den PKCS7-Dateien geschieht ebenfalls durch die Klasse `FileKeyEscrow`. Benötigt werden hierzu das Zertifikat des gesuchten Schlüssels (um die Datei zu finden) und der Recovery-Schlüssel (um die Datei zu entschlüsseln). Der Recovery-Schlüssel muß sich dabei in einem KeyStore befinden, der dem `FileKeyEscrow` übergeben wird. Das zugehörige Paßwort wird der Methode `recover()` mitgegeben.

```
escrow.setRecoveryKeys((KeyStore)ks);
Key a = escrow.recover((X509Certificate)userCert, password);
```

Kann die PKCS7-Datei zwar gefunden, aber nicht entschlüsselt werden (weil beispielsweise der Recovery-Schlüssel nicht im KeyStore vorhanden ist), so wirft die Methode eine `UndecryptableKeyException`. In dieser ist die unentschlüsselte PKCS7-Datei enthalten, die folglich weitergereicht werden kann. Auf diese Weise kann man zum Beispiel eine verteilte Entschlüsselung realisieren.

5.4 Applikationen

Die Programme werden mittels einer Batch-Datei (für Windows) oder einem Shell-Skript (für andere Plattformen) gestartet, wobei einige Parameter angegeben werden können.

5.4.1 Das Secret-Sharing-Kommandozeilen-Tool

Der FileSharer dient zum Zerteilen von beliebigen Dateien in n Anteile (die ebenfalls wieder in Dateien gespeichert werden) von denen t benötigt werden, um die Datei zurückzugewinnen. Dazu wird das XOR- oder das Shamir-Verfahren eingesetzt (Abschnitt 2.1 beziehungsweise Abschnitt 2.2.1). Die erzeugten Dateien können optional mit einem Paßwort versehen werden, um ihre Integrität zu schützen.

Datei aufteilen Durch Eingabe von

```
FileSharer --share <filename>
[-p <password>] [-t <threshold>] [-n <shares>] [--delete]
```

wird die Datei *filename* in *shares* Dateien *filename.1* bis *filename.n* aufgeteilt, von denen *threshold* beliebige vorhanden sein müssen, um die Datei zu rekonstruieren. Wenn mit der Option `-p` ein Passwort angegeben wird, werden die Anteile mit diesem Integritätspaßwort versehen. Wenn die Option `--delete` gesetzt ist, wird anschließend die Ursprungsdatei *filename* gelöscht. Die Defaulteinstellungen sehen ein 3-aus-5-Secret-Sharing vor.

Datei wiederherstellen Aus einer ausreichenden Anzahl von Anteilen kann die Datei *filename* mit

```
FileSharer --combine <filename> [-p <password>] [--delete]
```

rekonstruiert werden. Die Anteile werden in den Dateien *filename.1* bis *filename.n* erwartet. Das optionale Passwort *password* wird zur Überprüfung der Integrität der Anteile herangezogen. Wenn die Option `--delete` gesetzt ist, werden anschließend die Dateien mit den Anteilen gelöscht.

5.4.2 Das Key-Sharing-Kommandozeilen-Tool

Der KeySharer dient zur Erzeugung und Verwendung von verteilten RSA- und ElGamal-Schlüsseln. Er kann diese Schlüssel importieren oder erzeugen und anschließend auf mehrere Keystore-Dateien verteilen. Mit jeder dieser Keystore-Dateien können dann Teilsignaturen und Teilentschlüsselungen durchgeführt werden, welche dann ebenfalls mit KeySharer zusammengesetzt werden können.

Teilschlüssel erzeugen KeySharer kann RSA- und ElGamal-Schlüssel sowohl erzeugen als auch von einem Keystore importieren. Die daraus abgeleiteten Teilschlüssel speichert das Programm dann in eine Reihe abhängiger Keystores, welche an die Teilnehmer des Key-Sharing übermittelt werden können.

Der Befehl, um den Schlüssel *alias* aus dem Keystore *keystore* zu importieren und zu verteilen, lautet

```
KeySharer --share  [-t <threshold>] [-n <shares>]
--import <keystore> --key <alias> --password <password>
[--storetype <type>] [--storepass <storepass>].
```

Dabei wird das Password *password* verwendet, um den Schlüssel zu lesen, und das (optionale) Integritätspaßwort *storepass* um die Unversehrtheit des Eingabekeystores zu prüfen. Falls es sich dabei nicht um einen JavaKeyStore (JKS) handelt, kann dessen Typ mit der Option `--storetype` angegeben werden. Die Teilschlüssel erhalten in den jeweiligen Keystores ebenfalls den Namen *alias*. Die erzeugten Keystores werden als JavaKeyStores in den Dateien *keystore.1* bis *keystore.n* abgelegt. Falls diese Dateien bereits zuvor existierten, werden sie eingelesen und durch den neuen Schlüssel ergänzt. Die neuen Keystores werden ohne Integritätspaßwort gespeichert, der Schlüssel selbst wird durch das Paßwort *password* geschützt.

Ein neues Schlüsselpaar kann mit dem Befehl

```
KeySharer --create <alias> [-t <threshold>] [-n <shares>]
--keystore <keystore> --keytype <type> [--keysize <keysize>]
--password <password>
```

erzeugt und verteilt werden. Wie oben werden dabei die Teilschlüssel auf Keystores in Dateien *keystore.1* bis *keystore.n* verteilt. Sie werden dort unter dem Namen *alias* abgelegt und mit dem Paßwort *password* geschützt. Der Typ des Schlüssel kann entweder `RSA` oder `ElGamal` sein, die optionale *keysize* gibt die Schlüssellänge an.

Das verwendete Key-Sharing-Verfahren hängt von der Art des Schlüssels und von dem Bedarf nach Redundanz ab. RSA-Schlüssel werden nicht-redundant nach dem einfachen Verfahren aus Abschnitt 3.4 oder redundant nach dem Verfahren von Frankel, Gemmell, MacKenzie und Yung (Abschnitt 3.5.2) verteilt. ElGamal-Schlüssel werden nach den Verfahren aus Abschnitt 3.8 verteilt. Da eine redundante Verteilung von ElGamal-Schlüsseln Schlüssel spezieller Gestalt erfordert, steht diese Möglichkeit nur bei neu erzeugten Schlüsselpaaren zur Verfügung und nicht beim Import aus einem KeyStore.

Berechnungen mit den Teilschlüsseln durchführen Die Teilschlüssel können für Teilentschlüsselungen und (nur bei RSA) für Teilsignaturen verwendet werden. In beiden Fällen erzeugt KeySharer dabei eine Ausgabe, die später zusammen mit den Ausgaben der anderen Teilnehmer zur Gesamt-Entschlüsselung bzw. -Signatur kombiniert werden kann.

Bei der Entschlüsselung wird das PKCS7-Format unterstützt. Um eine verschlüsselte PKCS7-Datei (wie sie beispielsweise von der Klasse `KeyEscrow` erzeugt wird) teilweise zu entschlüsseln, wird der Befehl

```
KeySharer --decrypt <filename> --keystore <keystore> --password <password>
```

verwendet. Der bezeichnete Keystore wird dabei automatisch nach einem passenden Schlüssel durchsucht, der dann mit dem angegebenen Paßwort geladen werden kann. Die Teilentschlüsselung wird in die Datei *filename.n* geschrieben, wobei n die Nummer des Teilschlüssels ist.

Mit RSA-Teilschlüsseln können auch Teilsignaturen erzeugt werden. Hierbei werden rohe Signaturen sowie teilsignierte X509-Zertifikate und PKCS7-Dateien unterstützt.

```
KeySharer --sign <filename> --key <alias> --keystore <keystore>
--password <password> --sigAlg <algorithm>

KeySharer --signX509 <filename> --key <alias> --keystore <keystore>
--password <password> --signature <algorithm>

KeySharer --signPKCS7 <filename> --key <alias> --keystore <keystore>
--password <password> --signature <algorithm>
```

Teilberechnungen kombinieren

```
KeySharer --combine <filename>
```

Dateien verschlüsseln Der KeySharer kann auch dazu verwendet werden, um Dateien mit einem öffentlichen Schlüssel im PKCS7-Format zu verschlüsseln.

```
KeySharer --encrypt <filename> --cert <certfilename>
```

5.4.3 Das Key-Escrow-Kommandozeilen-Tool

Die Applikation KeyEscrow dient zum Verwahren von Schlüsseln in chiffrierten Dateien, zum Erzeugen von wiederherstellbaren Schlüsseln (Abschnitt 4.3) und zur Wiederherstellung der Schlüssel. Die Schlüssel werden dabei in Form von PKCS12-Dateien eingelesen und ausgegeben.

wiederherstellbaren Schlüssel erzeugen

```
KeyEscrow --create <filename> --keytype <keytype> --password <password>
--dname <dn> --cert <certfilename>
```

Der Schlüssel wird in eine PKCS12-Datei mit dem angegebenen Namen *filename* geschrieben und mit dem Password *password* geschützt. In dieser enthalten ist auch ein Dummy-X.509-Zertifikat mit dem öffentlichen Schlüssel. Der Name des Schlüsselinhabers kann mit der Option `--dname` gesetzt werden, wobei die X.501-Namenskonventionen eingehalten werden müssen (zum Beispiel `CN=Thilo Planz, O=TU Darmstadt, C=DE`).

wiederherstellbaren Schlüssel rekonstruieren

```
KeyEscrow --recover <filename> --cert <certfilename> --key <alias>
--keystore <keystore> --password <password>
```

Ein wiederherstellbarer Schlüssel kann aus dem öffentlichen Schlüssel, der einem X.509-Zertifikat entnommen wird, rekonstruiert werden. Der rekonstruierte Schlüssel wird dann in die PKCS12-Datei *filename* geschrieben. Zur Rekonstruktion ist allerdings ein Recovery-Key notwendig, der dem Keystore *keystore* entnommen wird (unter dem Namen *alias* und Paßwort *password*).

Schlüssel in einer Datei verwahren

```
KeyEscrow --escrow <filename> --password <password>
--escrowDir <escrowdir> --cert <certfilename>
```

KeyEscrow legt im bezeichneten Verzeichnis *escrowDir* verschlüsselte Benutzerschlüssel ab (im PKCS7-Format). Der Recovery-Operator (der die Dateien wieder entschlüsseln kann) wird durch sein Zertifikat in der Datei *certfilename* bezeichnet. Der Benutzerschlüssel selber wird der PKCS12-Datei *filename* entnommen, die sich mit dem Paßwort *password* öffnen lassen muß.

Schlüssel aus der Datei wiederherstellen

```
KeyEscrow --recover <filename> --cert <certfilename>
--escrowDir <escrowdir> --keystore <keystore> --password <password>
```

Der rekonstruierte Schlüssel wird in eine mit dem angegeben Password geschützte PKCS12-Datei *filename* geschrieben. Der gewünschte Schlüssel wird anhand des zugehörigen Zertifikats in der Datei *certfilename* identifziert. Zur Rekonstruktion ist ein Recovery-Key notwendig, der dem Keystore *keystore* entnommen wird (er wird automatisch gesucht und mit dem Paßwort *password* geladen). Wenn es sich dabei nur um Teilschlüssel handelt, wird statt der PKCS12-Datei der partiell dechiffrierte Schlüssel ausgegeben. Die so entstehenden Teile können mit dem KeySharer zusammengesetzt werden. Wenn der Recovery-Schlüssel überhaupt nicht vorliegt, wird statt dessen die verschlüsselte PKCS7-Datei ausgegeben. Diese kann dann weitergereicht werden, beispielsweise im Rahmen eines verteilten Recovery mit der Anwendung KeySharer.

Kapitel 6

Ausblick

Wir haben mit dieser Arbeit einen Einstieg vor allem in Techniken des Secret-Sharing und der Threshold Cryptography geben wollen. Außerdem haben wir uns mit der Key-Escrow-Problematik und einigen Ansätzen hierzu beschäftigt. Durch die Implementierung eines großen Teils der genannten Algorithmen konnten wir zum einen deren Einsatzfähigkeit demonstrieren und zum anderen einen hoffentlich nützlichen Beitrag zum FlexiTrust-Projekt des Fachgebietes leisten.

Entsprechend unserer ursprünglichen Motivation, nämlich der sicheren Speicherung privater Schlüssel, haben wir uns auf eine bestimmte Modellierung eingeschränkt, so daß viele andere interessante Verfahren der Threshold Cryptography und ähnlicher Gebiete unbeachtet geblieben sind. Dies lag sicher auch am begrenzten Umfang dieser Arbeit. Wir wollen daher abschließend noch ein paar Stichworte für eine weitergehende Beschäftigung geben.

Zunächst einmal könnte man unsere Implementierung noch abrunden. Einige der geschilderten Verfahren sind nicht (Pedersen-Verfahren) oder nur teilweise (Shoup-Verfahren ohne Verifikation) implementiert. Auch ist die Implementierung nicht auf Performanz ausgerichtet und bietet durch die Kommandozeilenapplikationen und die Programmierschnittstelle eine zwar brauchbare, aber nur sehr rudimentäre Infrastruktur. Interessant wäre ihr Einsatz in einem größeren Projekt für eine verteilte kryptographische Applikation, etwa eine verteilte CA.

Ein sehr bedeutsames Gebiet, das wir nur am Rande gestreift haben, ist die verteilte Schlüsselerzeugung. In konsequenter Weiterführung des Modells, in dem es gilt, den Schlüssel vor unvertrauenswürdigen Parteien zu schützen, wird hier auch die Existenz eines vertrauenswürdigen Gebers abgestritten, der einen Schlüssel unbeaufsichtigt erzeugen könnte. An seine Stelle tritt eine Kooperation von Teilnehmern, die sich alle nicht vollständig vertrauen, aber trotzdem gemeinsam einen Schlüssel erzeugen und verwenden können.

Ebenfalls sinnvoll wäre die Implementierung weiterer Algorithmen, etwa einer verteilten Signatur gemäß DSS, die nähere Erforschung und Beurteilung der kleptographischen Methoden zur Schlüsselwiederherstellung sowie die proaktive Erneuerung der Anteile durch die Teilnehmer selbst.

Anhang A

Abkürzungsverzeichnis

ASN.1 *abstract syntax notation.*
eine Beschreibungssprache für Datenstrukturen. Wird unter anderem bei X.509 eingesetzt. ASN.1-Daten werden im Gegensatz zu XML-Daten in einem kompakten Binärformat gespeichert.

CA *certification authority.*
Zertifizierungsinstanz, Trustcenter. Vertrauenswürdige Stelle, die die eindeutige Zuordnung von öffentlichen Schlüsseln zu den Mitgliedern der Kommunikationsgemeinschaft garantiert.

DN *distinguished name.*
Nach dem X.500-Standard für Verzeichnisdienste verfügt jeder Teilnehmer über einen eindeutigen Namen, der aus mehreren Namensbestandteilen (z.B. Organisationsname, Ländername, Name der Person) aufgebaut ist, zwischen denen eine Hierarchie besteht. Auch die Teilnehmer einer X.509-PKI werden durch ihren DN identifiziert.

JCA *Java Cryptography Architecture.*
Teil der Java-Plattform, der den Zugriff auf kryptographische Algorithmen und Daten regelt

PKCS *public key cryptography standards.*
von den RSA-Laboratories entwickelte Familie von Standards. PKCS#7 definiert ein Format für signierte und/oder verschlüsselte Dateien, PKCS#8 ein Binärformat für private Schlüssel und PKCS#12 das sogenannte Softtoken, ein geschütztes Transportformat für Schlüsselpaare.

PKI *Public-Key-Infrastruktur.*
System aus Richtlinien, Abläufen, Institutionen und Datenformaten zur Verwaltung der in der asymmetrischen Kryptographie benötigten öffentlichen Schlüssel

SSL *secure socket layer.*
Protokoll zur sicheren Kommunikation über das Internet. Server und optional Client authentifizieren sich mit Zertifikaten, die übertragenen Daten werden mit einem Einmalschlüssel verschlüsselt.

X.509 *ITU Empfehlung X.509, auch ISO/IEC 9594-8.*
: dominierender PKI-Standard. Enthält unter anderem ein flexibles Zertifikatsformat, das mittlerweile in fast allen Anwendungen verwendet wird.

XML *extensible markup language.*
: universelles Austauschformat für strukturierte Daten. XML zeichnet sich gegenüber dem älteren SGML durch eine stark vereinfachte Syntax aus, die die Entwicklung vom XML-fähigen Anwendungen vereinfacht hat. XML hat in letzter Zeit eine große Verbreitung erreicht. XML ist im Gegensatz zu ASN.1 kein Binärformat, sondern textbasiert.

Anhang B

Symbolverzeichnis

Secret-Sharing

$f(x)$	Beim Shamir-Verfahren das Polynom über $(\mathbb{Z}/p\mathbb{Z})$, mit dessen Hilfe das Geheimnis $s = f(0)$ verteilt wird.
$\lambda_{i,\Lambda}$	Interpolationskoeffizient für Teilnehmer i aus Λ: $\lambda_{i,\Lambda} = \prod_{l \in \Lambda \setminus \{i\}} \frac{l}{l-i}$.
Λ	Menge der Teilnehmer(-nummern) für die Rekonstruktion
n	Anzahl der Anteile, in die das Geheimnis aufgeteilt wird.
$\binom{t}{n}$	t-aus-n-Secret-Sharing
p	Beim Shamir-Verfahren die Primzahl, die den Raum der Geheimnisse $(\mathbb{Z}/p\mathbb{Z})$bestimmt.
s	das Geheimnis. Beim Shamir-Verfahren $s \in (\mathbb{Z}/p\mathbb{Z})$
s_i	die Teilgeheimnisse. Jeder Teilnehmer i erhält ein s_i.
t	*Threshold.* Anzahl der benötigten Anteile, um das Geheimnis zu rekonstruieren.

RSA

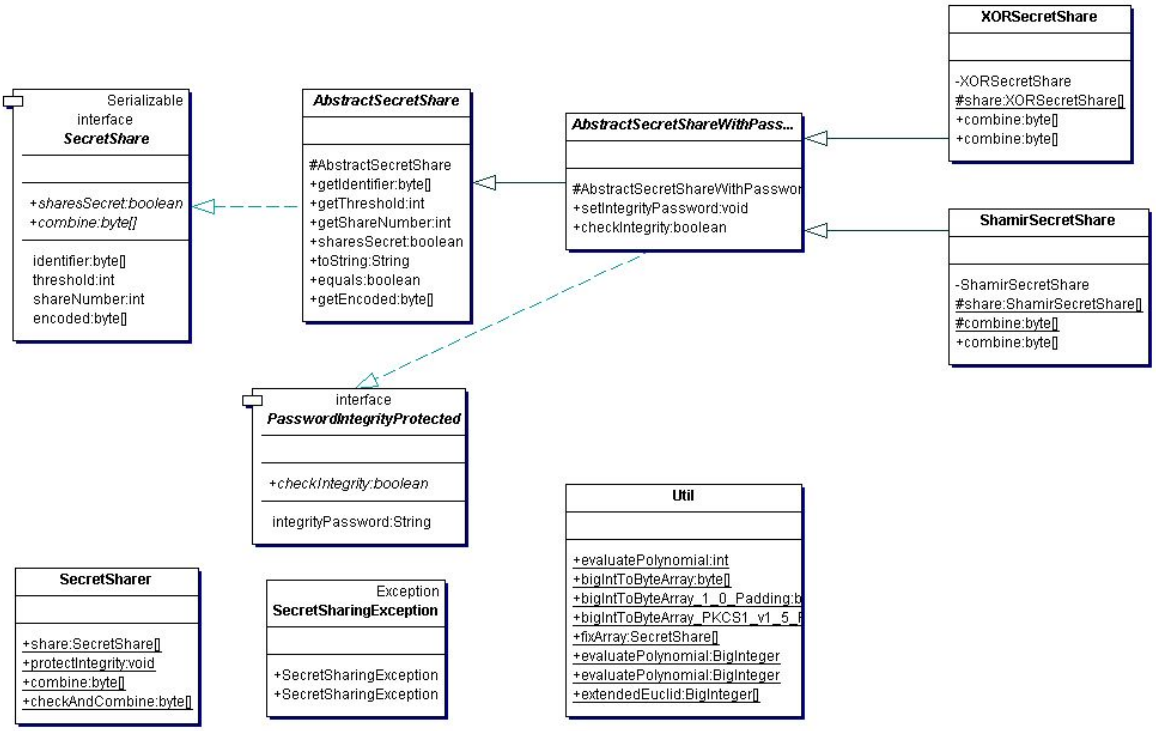
c	Schlüsseltext, verschlüsselte Nachricht $c = m^e \pmod N$
d	privater Exponent, wird zum Entschlüsseln und Signieren verwendet.
e	öffentlicher Exponent, wird zum Verschlüsseln und Verifizieren verwendet.
m	Klartext, Nachricht, Zahl aus $(\mathbb{Z}/N\mathbb{Z})$, bei Signaturen ein Hashwert
N	öffentlicher Modulus $N = pq$
p	geheimer Primfaktor des Modulus $N = pq$
q	geheimer Primfaktor des Modulus $N = pq$
s	Signatur $s = m^d \pmod N$

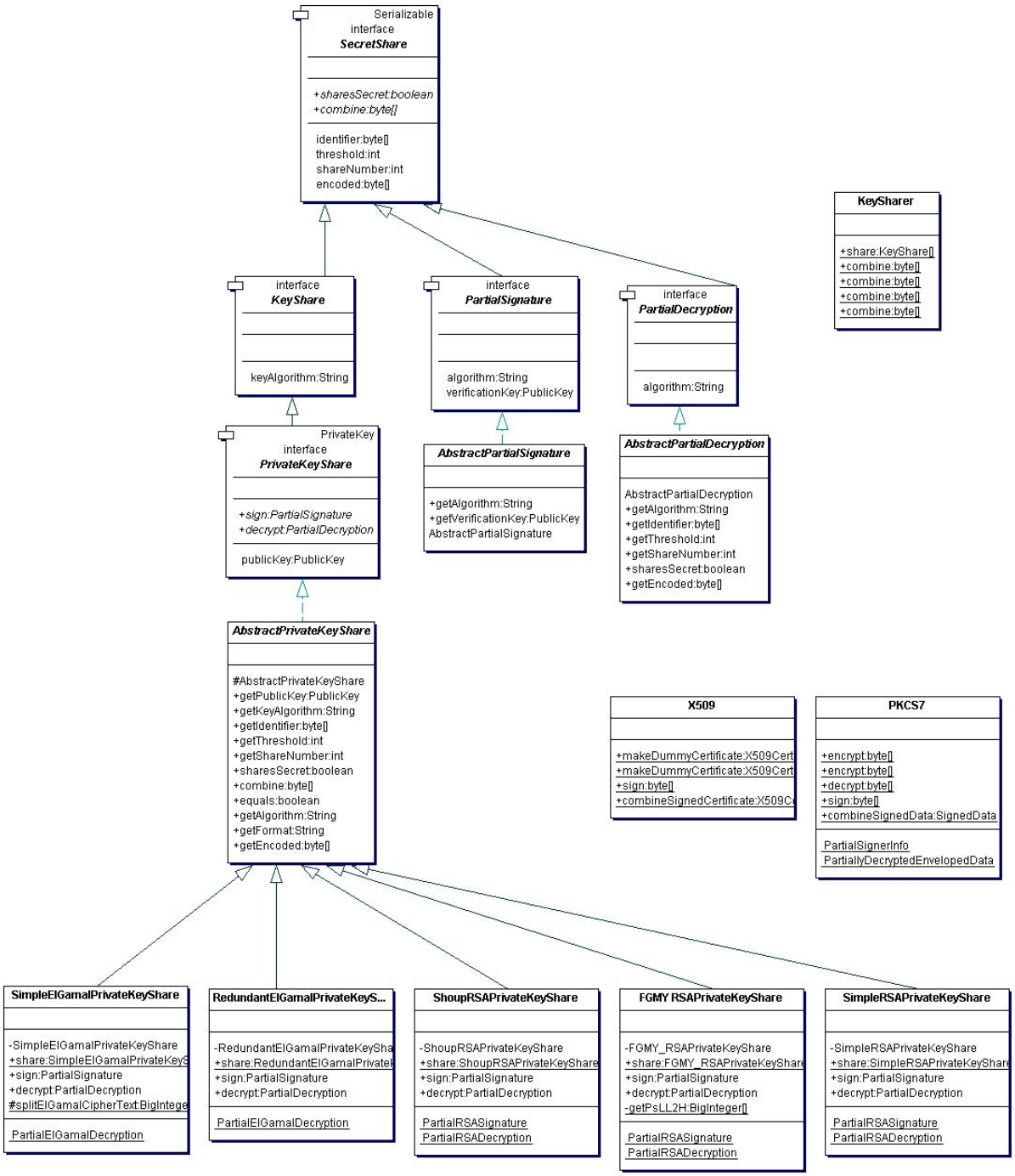
ElGamal

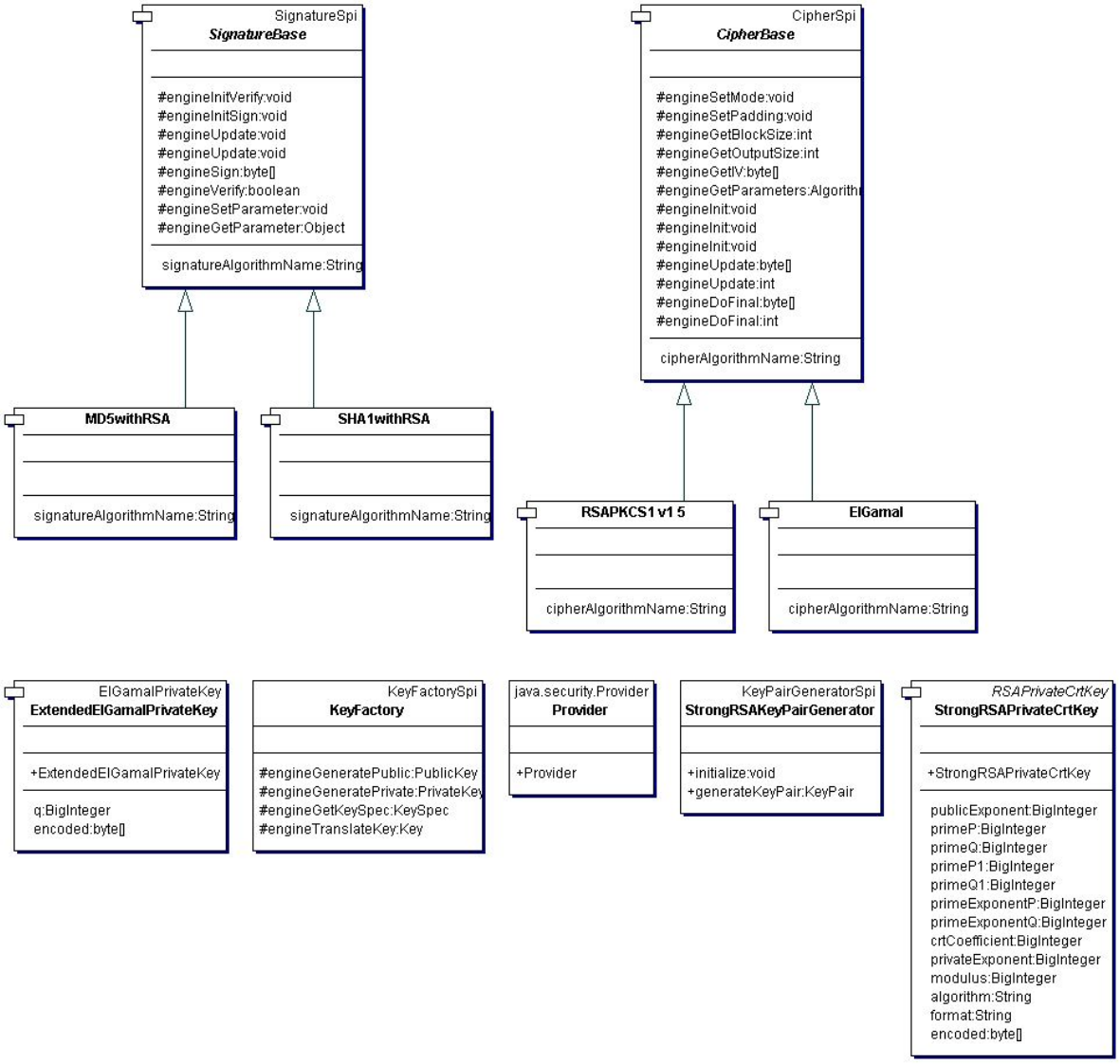
- a privater Exponent, wird zum Entschlüsseln und Signieren verwendet.
- A öffentlicher Schlüssel, wird zum Verschlüsseln und Verifizieren verwendet. $A = g^a \pmod{p}$
- B Teil das Schlüsseltexts
- g Basiselement, Teil des öffentlichen Schlüssels.
- p öffentlich bekannte Primzahl, bestimmt die Gruppe $(\mathbb{Z}/p\mathbb{Z})$, in der gerechnet wird.
- q optionale öffentliche Primzahl, die bewirkt, daß ElGamal-Exponenten nicht modulo $p-1$, sondern modulo q gerechnet werden. Wird im DSA-Verfahren verwendet, und bei ElGamal-Key-Sharing. $p = mq + 1$

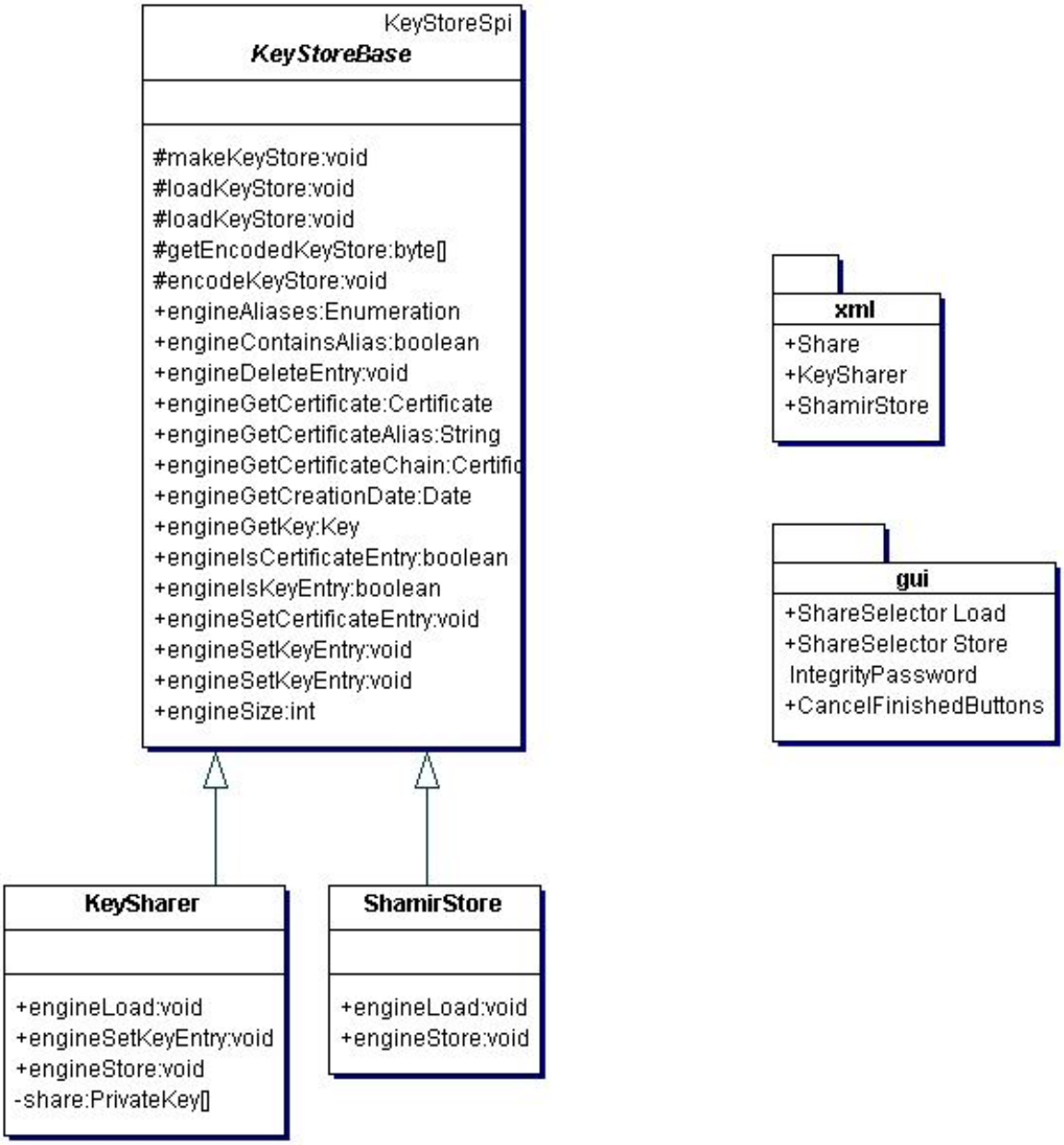
Anhang C

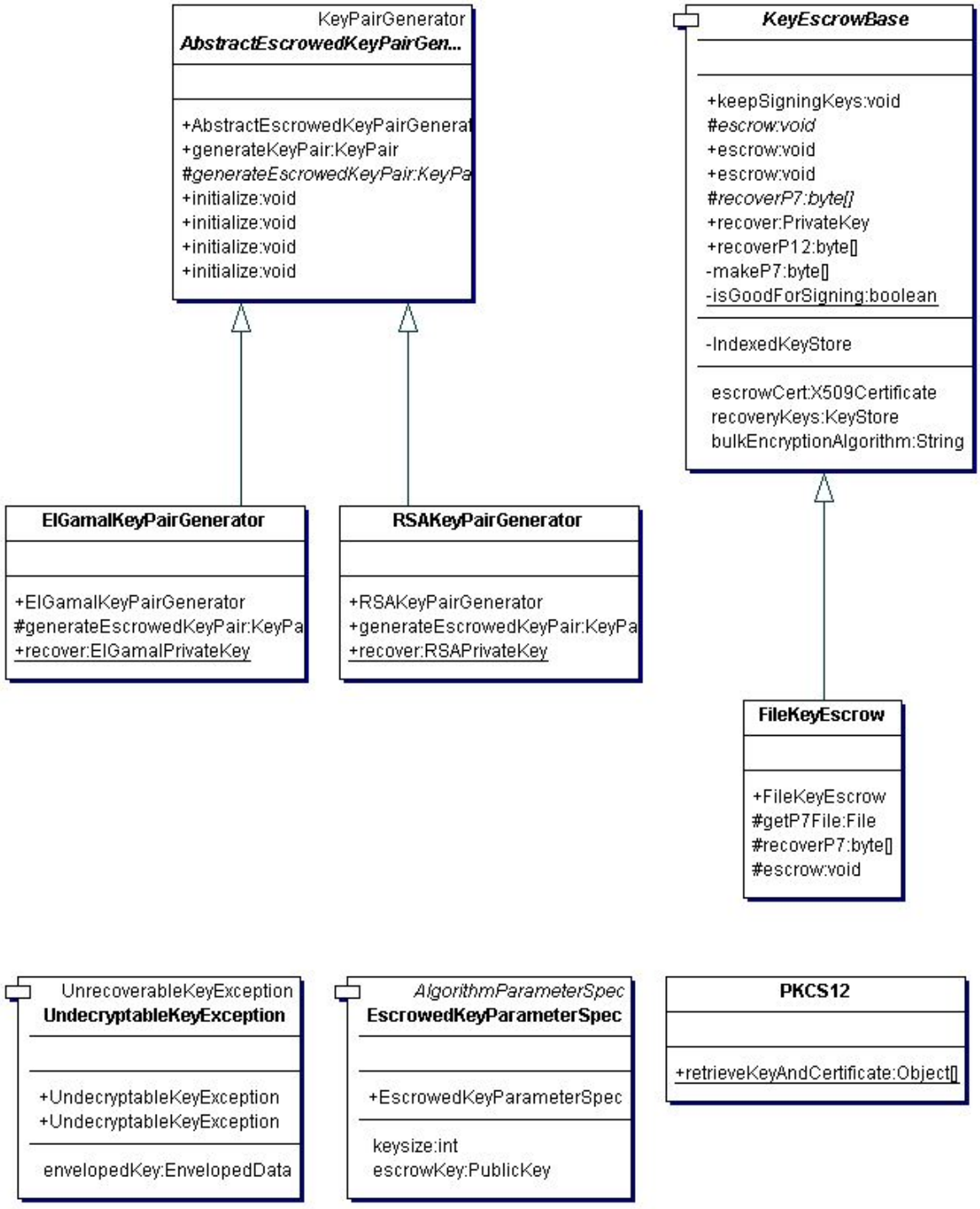
UML-Klassendiagramme

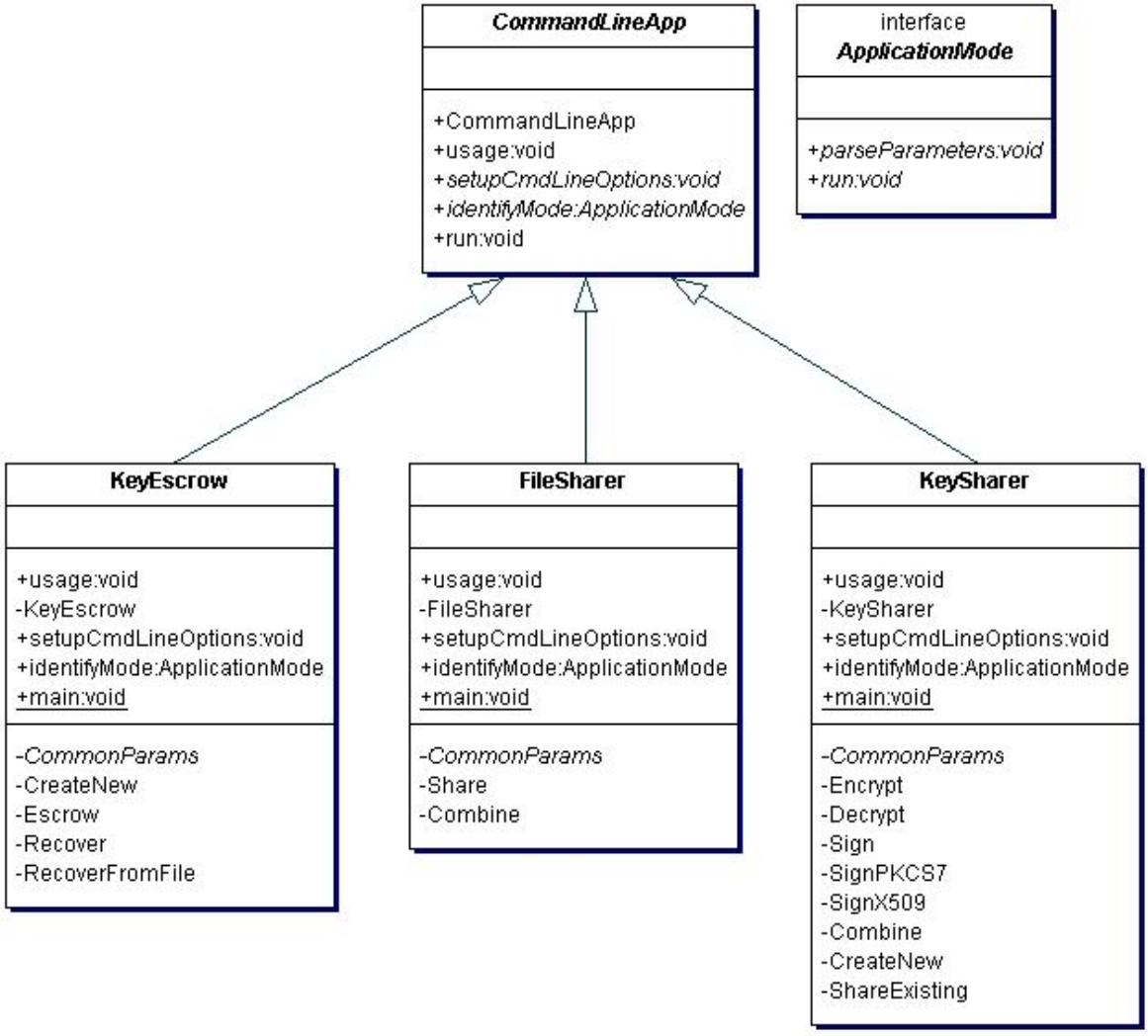



Serializable
interface
SecretShare
+sharesSecret:boolean
+combine:byte[]
identifier:byte[]
threshold:int
shareNumber:int
encoded:byte[]
KeySharer
+share:KeyShare[]
+combine:byte[]
+combine:byte[]
+combine:byte[]
+combine:byte[]
interface
KeyShare
keyAlgorithm:String
interface
PartialSignature
algorithm:String
verificationKey:PublicKey
interface
PartialDecryption
algorithm:String
PrivateKey
interface
PrivateKeyShare
+sign:PartialSignature
+decrypt:PartialDecryption
publicKey:PublicKey
AbstractPartialSignature
+getAlgorithm:String
+getVerificationKey:PublicKey
AbstractPartialSignature
AbstractPartialDecryption
AbstractPartialDecryption
+getAlgorithm:String
+getIdentifier:byte[]
+getThreshold:int
+getShareNumber:int
+sharesSecret:boolean
+getEncoded:byte[]
AbstractPrivateKeyShare
#AbstractPrivateKeyShare
+getPublicKey:PublicKey
+getKeyAlgorithm:String
+getIdentifier:byte[]
+getThreshold:int
+getShareNumber:int
+sharesSecret:boolean
+combine:byte[]
+equals:boolean
+getAlgorithm:String
+getFormat:String
+getEncoded:byte[]
X509
+makeDummyCertificate:X509Cert
+makeDummyCertificate:X509Cert
+sign:byte[]
+combineSignedCertificate:X509C
PKCS7
+encrypt:byte[]
+encrypt:byte[]
+decrypt:byte[]
+sign:byte[]
+combineSignedData:SignedData
PartialSignerInfo
PartiallyDecryptedEnvelopedData
SimpleElGamalPrivateKeyShare
-SimpleElGamalPrivateKeyShare
+share:SimpleElGamalPrivateKeyS
+sign:PartialSignature
+decrypt:PartialDecryption
#splitElGamalCipherText:BigInteger
PartialElGamalDecryption
RedundantElGamalPrivateKeyS...
-RedundantElGamalPrivateKeySha
+share:RedundantElGamalPrivateK
+sign:PartialSignature
+decrypt:PartialDecryption
PartialElGamalDecryption
ShoupRSAPrivateKeyShare
-ShoupRSAPrivateKeyShare
+share:ShoupRSAPrivateKeyShare
+sign:PartialSignature
+decrypt:PartialDecryption
PartialRSASignature
PartialRSADecryption
FGMY RSAPrivateKeyShare
-FGMY_RSAPrivateKeyShare
+share:FGMY_RSAPrivateKeyShare
+sign:PartialSignature
+decrypt:PartialDecryption
-getPsLL2H:BigInteger[]
PartialRSASignature
PartialRSADecryption
SimpleRSAPrivateKeyShare
-SimpleRSAPrivateKeyShare
+share:SimpleRSAPrivateKeyShare
+sign:PartialSignature
+decrypt:PartialDecryption
PartialRSASignature
PartialRSADecryption


SignatureSpi
SignatureBase
#engineInitVerify:void
#engineInitSign:void
#engineUpdate:void
#engineUpdate:void
#engineSign:byte[]
#engineVerify:boolean
#engineSetParameter:void
#engineGetParameter:Object
signatureAlgorithmName:String
CipherSpi
CipherBase
#engineSetMode:void
#engineSetPadding:void
#engineGetBlockSize:int
#engineGetOutputSize:int
#engineGetIV:byte[]
#engineGetParameters:Algorith
#engineInit:void
#engineInit:void
#engineInit:void
#engineUpdate:byte[]
#engineUpdate:int
#engineDoFinal:byte[]
#engineDoFinal:int
cipherAlgorithmName:String
MD5withRSA
signatureAlgorithmName:String
SHA1withRSA
signatureAlgorithmName:String
RSAPKCS1 v1 5
cipherAlgorithmName:String
ElGamal
cipherAlgorithmName:String
ElGamalPrivateKey
ExtendedElGamalPrivateKey
+ExtendedElGamalPrivateKey
q:BigInteger
encoded:byte[]
KeyFactorySpi
KeyFactory
#engineGeneratePublic:PublicKey
#engineGeneratePrivate:PrivateKey
#engineGetKeySpec:KeySpec
#engineTranslateKey:Key
java.security.Provider
Provider
+Provider
KeyPairGeneratorSpi
StrongRSAKeyPairGenerator
+initialize:void
+generateKeyPair:KeyPair
RSAPrivateCrtKey
StrongRSAPrivateCrtKey
+StrongRSAPrivateCrtKey
publicExponent:BigInteger
primeP:BigInteger
primeQ:BigInteger
primeP1:BigInteger
primeQ1:BigInteger
primeExponentP:BigInteger
primeExponentQ:BigInteger
crtCoefficient:BigInteger
privateExponent:BigInteger
modulus:BigInteger
algorithm:String
format:String
encoded:byte[]


KeyStoreSpi
KeyStoreBase
#makeKeyStore:void
#loadKeyStore:void
#loadKeyStore:void
#getEncodedKeyStore:byte[]
#encodeKeyStore:void
+engineAliases:Enumeration
+engineContainsAlias:boolean
+engineDeleteEntry:void
+engineGetCertificate:Certificate
+engineGetCertificateAlias:String
+engineGetCertificateChain:Certific
+engineGetCreationDate:Date
+engineGetKey:Key
+engineIsCertificateEntry:boolean
+engineIsKeyEntry:boolean
+engineSetCertificateEntry:void
+engineSetKeyEntry:void
+engineSetKeyEntry:void
+engineSize:int
xml
+Share
+KeySharer
+ShamirStore
gui
+ShareSelector Load
+ShareSelector Store
IntegrityPassword
+CancelFinishedButtons
KeySharer
+engineLoad:void
+engineSetKeyEntry:void
+engineStore:void
-share:PrivateKey[]
ShamirStore
+engineLoad:void
+engineStore:void


KeyPairGenerator
AbstractEscrowedKeyPairGen...
+AbstractEscrowedKeyPairGenerat
+generateKeyPair:KeyPair
#generateEscrowedKeyPair:KeyPa
+initialize:void
+initialize:void
+initialize:void
+initialize:void
KeyEscrowBase
+keepSigningKeys:void
#escrow:void
+escrow:void
+escrow:void
#recoverP7:byte[]
+recover:PrivateKey
+recoverP12:byte[]
-makeP7:byte[]
-isGoodForSigning:boolean
-IndexedKeyStore
escrowCert:X509Certificate
recoveryKeys:KeyStore
bulkEncryptionAlgorithm:String
ElGamalKeyPairGenerator
+ElGamalKeyPairGenerator
#generateEscrowedKeyPair:KeyPa
+recover:ElGamalPrivateKey
RSAKeyPairGenerator
+RSAKeyPairGenerator
+generateEscrowedKeyPair:KeyPa
+recover:RSAPrivateKey
FileKeyEscrow
+FileKeyEscrow
#getP7File:File
#recoverP7:byte[]
#escrow:void
UnrecoverableKeyException
UndecryptableKeyException
+UndecryptableKeyException
+UndecryptableKeyException
envelopedKey:EnvelopedData
AlgorithmParameterSpec
EscrowedKeyParameterSpec
+EscrowedKeyParameterSpec
keysize:int
escrowKey:PublicKey
PKCS12
+retrieveKeyAndCertificate:Object[]


CommandLineApp
+CommandLineApp
+usage:void
+setupCmdLineOptions:void
+identifyMode:ApplicationMode
+run:void
interface
ApplicationMode
+parseParameters:void
+run:void
KeyEscrow
+usage:void
-KeyEscrow
+setupCmdLineOptions:void
+identifyMode:ApplicationMode
+main:void
-CommonParams
-CreateNew
-Escrow
-Recover
-RecoverFromFile
FileSharer
+usage:void
-FileSharer
+setupCmdLineOptions:void
+identifyMode:ApplicationMode
+main:void
-CommonParams
-Share
-Combine
KeySharer
+usage:void
-KeySharer
+setupCmdLineOptions:void
+identifyMode:ApplicationMode
+main:void
-CommonParams
-Encrypt
-Decrypt
-Sign
-SignPKCS7
-SignX509
-Combine
-CreateNew
-ShareExisting

Anhang D

Benutzerhandbuch Kommandozeilenapplikationen

D.1 FileSharer

Datei aufteilen

```
FileSharer --share <filename>
[-p <password>] [-t <threshold>] [-n <shares>] [--delete]
```

Datei wiederherstellen

```
FileSharer --combine <filename> [-p <password>] [--delete]
```

D.1.1 KeySharer

Teilschlüssel erzeugen

```
KeySharer --share  [-t <threshold>] [-n <shares>]
--import <keystore> --key <alias> --password <password>
[--storetype <type>] [--storepass <storepass>].
```

```
KeySharer --create <alias> [-t <threshold>] [-n <shares>]
--keystore <keystore> --keytype <type> [--keysize <keysize>]
--password <password>
```

Berechnungen mit den Teilschlüsseln durchführen

```
KeySharer --decrpyt <filename> --keystore <keystore> --password <password>
```

```
KeySharer --sign <filename> --key <alias> --keystore <keystore>
--password <password> --sigAlg <algorithm>
```

```
KeySharer --signX509 <filename> --key <alias> --keystore <keystore>
--password <password> --signature <algorithm>
```

```
KeySharer --signPKCS7 <filename> --key <alias> --keystore <keystore>
--password <password> --signature <algorithm>
```

Teilberechnungen kombinieren

```
KeySharer --combine <filename>
```

Dateien verschlüsseln

```
KeySharer --encrypt <filename> --cert <certfilename>
```

D.2 KeyEscrow

wiederherstellbaren Schlüssel erzeugen

```
KeyEscrow --create <filename> --keytype <keytype> --password <password>
--dname <dn> --cert <certfilename>
```

wiederherstellbaren Schlüssel rekonstruieren

```
KeyEscrow --recover <filename> --cert <certfilename> --key <alias>
--keystore <keystore> --password <password>
```

Schlüssel in einer Datei verwahren

```
KeyEscrow --escrow <filename> --password <password>
--escrowDir <escrowdir> --cert <certfilename> :
```

Schlüssel aus der Datei wiederherstellen

```
KeyEscrow --recover <filename> --cert <certfilename>
--escrowDir <escrowdir> --key <alias> --keystore <keystore> --password <password>
```

Anhang E

Benutzerhandbuch Programmierschnittstelle

E.1 Secret-Sharing-Basisfunktionalität

Geheimnis verteilen

```
public static SecretShare[] share
(byte[] secret, int threshold, int sharenumber)
throws InvalidParameterException, NoSuchAlgorithmException
```

Integritätspaßwort setzen

```
public static void protectIntegrity
(PasswordIntegrityProtected[] shares, String password)
throws NoSuchAlgorithmException
```

Anteile serialisieren

```
byte[] bytes = share.getEncoded();
```

Anteile zusammensetzen

```
public static byte[] combine
(SecretShare[] shares)
throws NoSuchAlgorithmException, SecretSharingException

public static byte[] checkAndCombine
(SecretShare[] shares, String integrityPassword)
throws NoSuchAlgorithmException, SecretSharingException
```

E.2 Key-Sharing-Basisfunktionalität

Schlüssel verteilen

```
public static KeyShare[] share
(Key secret, int threshold, int sharenumber)
throws InvalidParameterException, NoSuchAlgorithmException, InvalidKeySpecException
```

Teilsignaturen und Teilentschlüsselungen erstellen

```
Signature signer = Signature.getInstance("DistributedMD5withRSA");
signer.initSign(keyshare);
signer.update(message);
byte[] partialSignature = signer.sign();
```

Teilberechnungen zusammenführen

```
public static byte[] combine
(byte[][] shares, byte[] message)
throws NoSuchAlgorithmException, SecretSharingException
```

E.3 Der Shamir-KeyStore

Konfiguration

```
<shamir-store threshold="2" load="AUTO store="AUTO" integrity-password="Passwort">
        <share file="1a.share" />
        <share file="2b.share" />
        <share file="3c.share" />
</shamir-store>
```

Instanz erzeugen

```
KeyStore ks = KeyStore.getInstance("ShamirStore");
```

Keystore laden

```
FileInputStream in = new FileInputStream("configuration.xml");
ks.load(in, password);
in.close();
```

Keystore verwenden

```
ks.setKeyEntry(alias, key, password, certificateChain);
Key key = ks.getKey(alias, password);
Certificate cert = ks.getCertificate(alias);
```

Keystore speichern

```
ks.store(null,  password);
```

E.4 Der KeySharer-KeyStore

Konfiguration

```
<key-sharer threshold="2" >
        <share file="1a.keystore" />
        <share file="2b.keystore" />
        <share file="3c.keystore" />
</key-sharer>
```

Instanz erzeugen

```
KeyStore ks = KeyStore.getInstance("KeySharer");
```

Keystore laden

```
FileInputStream in = new FileInputStream("configuration.xml");
ks.load(in, password);
in.close();
```

Keystore verwenden

```
ks.setKeyEntry(alias, key, password, certificateChain);
```

Keystore speichern

```
ks.store(null,  password);
```

E.5 Das Key-Escrow-System

Wiederherstellbares Schlüsselpaar erzeugen

```
KeyPairGenerator keygen = KeyPairGenerator.getInstance("RSA", "KeySharingProvider");
keygen.initialize(new EscrowedKeyParameterSpec(1024, (PublicKey)escrow));
KeyPair pair = keygen.generateKeyPair();
```

privaten Schlüssel wiederherstellen

```
PrivateKey recovered = RSAKeyPairGenerator.recover(
(RSAPublicKey)key, (PrivateKey)recovery, (PublicKey)escrow);
```

privaten Schlüssel im Dateisystem verwahren

```
FileKeyEscrow escrow = new FileKeyEscrow((File)escrowDir);
escrow.setEscrowCert((X509Certificate)cert);
escrow.escrow((Key)key, (X509Certificate)userCert);
```

privaten Schlüssel aus dem Dateisystem wiederherstellen

```
escrow.setRecoveryKeys((KeyStore)ks);
Key a = escrow.recover((X509Certificate)userCert, password);
```

Index

AbstractEscrowedKeyPairGenerator, 60
AbstractPartialDecryption, 53
AbstractPartialSignature, 53
AbstractPrivateKeyShare, 53
AbstractSecretShare, 50
AbstractSecretShareWithPassword, 50
access structures, 14
ApplicationMode, 62
AppTests, 64
ASN.1, 77
auto-escrowing keys, 40

CA, 77
CancelFinishedButton, 59
CipherBase, 56
CommandLineApp, 62

Diffie-Hellman-Problem, 29
distinguished name, 77
DN, 77
Double Dispatch, 51
DSA, 31

ElGamal, 29
    Sicherheit des Verfahrens, 29
ElGamal, 56
ElGamalKeyPairGenerator, 60
EscrowedKeyParameterSpec, 60
ExtendedElGamalPrivateKey, 56

FGMY_RSAPrivateKeyShare, 53
FileKeyEscrow, 60
FileSharer, 71
FileSharer, 62
Function-Sharing, 14

Integritätsschutz, 51
IntegrityPassword, 59

Java Cryptography Architecture, 44
JavaDoc, 44
JCA, 44, 77

Key Escrow, 35
Key Recovery, 35
Key-Sharing, 16
    ElGamal, 31
    Modell, 17
    RSA, 21, 22, 27
KeyEscrow, 74
KeyEscrow, 62
KeyEscrowBase, 60
KeyEscrowTests, 64
KeyFactory, 56
KeyShare, 51
KeySharer, 72
KeySharer, 54, 57, 62, 68
KeySharerTests, 64
KeyStoreBase, 57
Kleptographie, 39
Kryptographie
    asymmetrisch, 16
    symmetrisch, 8

Lagrange, 10

MD5withRSA, 56

One-Time-Pad-Verschlüsselung, 8

Padding, 21
PartialDecryption, 53
PartialSignature, 53
PasswordIntegrityProtected, 50
PKCS, 77
PKCS12, 62
PKCS7, 54
PKCS7Tests, 64
PKCS8, 54

PKI, 77
Polynominterpolation, 10
Pretty Awful Privacy, 40
PrivateKeyShare, 53
Provider, 54
Public-Key-Kryptographie, 16

RedundantElGamalPrivateKeyShare, 54
RSA, 19
  Assumption, 19
  Sicherheit des Verfahrens, 19
RSAKeyPairGenerator, 60
RSAPKCS1_v1_5, 56

Schlüsselerzeugung
  ElGamal, 30
    Teilschlüssel, 31
    wiederherstellbar, 43
  RSA, 20
    Teilschlüssel, 21, 23, 26, 27
    wiederherstellbar, 40
  verteilt, 32
Schlüsselverwendung
  ElGamal, 30
    verteilt, 32
  RSA, 20
    verteilt, 22, 24, 26, 28
Secret-Sharing, 7
  ideal, 14
  ohne Geber, 14
  Pedersen-Verfahren, 11
  perfekt, 13
  proaktiv, 14
  redundant, 9, 14
  robust, 14
  Shamir-Verfahren, 10
    blockweise, 50
  verifizierbar, 11
  XOR-Methode, 9
SecretShare, 49
SecretSharer, 50
SecretSharerTests, 64
SecretSharingException, 50
SETUP, 39
SHA1withRSA, 56
ShamirSecretShare, 50
ShamirStore, 57, 67
ShamirStoreTests, 64
Shannon, 8
Share, 57
ShareSelector_Load, 59
ShareSelector_Store, 59
Shoup, 27
ShoupRSAPrivateKeyShare, 53
Sicherheit
  berechnungssicher, 8
  informationstheoretisch, 8
SignatureBase, 56
SimpleElGamalPrivateKeyShare, 53
SimpleRSAPrivateKeyShare, 53
SSL, 77
StrongRSAKeyPairGenerator, 56
StrongRSAPrivateCrtKey, 56

Threshold Cryptography, 18

UndecryptableKeyException, 60
Util, 50

wiederherstellbare Schlüssel, 39

X.509, 77
X509, 54
X509Tests, 64
XML, 78
XORSecretShare, 50

Literaturverzeichnis

[AAB+97] Hal Abelson, Ross Anderson, Steven M. Bellovin, Josh Benaloh, Matt Blaze, Whitfield Diffie, John Gilmore, Peter G. Neumann, Ronald L. Rivest, Jeffrey I. Schiller und Bruce Schneier: *The risks of key recovery, key escrow & trusted third party encryption.* `http://www.cdt.org/crypto/risks98/`, 1997. A report by an ad hoc group of cryptographers and computer scientists.

[ABF+99] Helo Appel, Ingrid Biehl, Arnulph Fuhrmann, Markus Ruppert, Tsuyoshi Takagi, Akira Takura und Christian Valentin: *Ein sicherer, robuster Zeitstempeldienst auf der Basis verteilter RSA-Signaturen.* Technischer Report TI-22/99, Fachgebiet Theoretische Informatik, TU Darmstadt, 1999.

[BF97] Dan Boneh und M. Franklin: *Efficient generation of shared RSA keys.* Advances in Cryptology–CRYPTO'97, S. 425–439, 1997.

[Bla79] G. R. Blakley: *Safeguarding cryptographic keys.* In *Proc. AFIPS 1979 National Computer Conference*, S. 313–317. AFIPS, 1979.

[CKLW00] Ingmar Camphausen, Stefan Kelm, Britta Liedtke und Lars Weber: *Aufbau und Betrieb einer Zertifizierungsinstanz.* Deutsches Forschungsnetz, DFN-PCA – Vogt-Kölln-Straße 30 – 22527 Hamburg, März 2000.

[DF90] Yvo Desmedt und Yair Frankel: *Threshold cryptosystems.* Advances in Cryptology–CRYPTO'89, 435:307–315, 1990.

[DH76] Whitfield Diffie und Martin E. Hellman: *New Directions in Cryptography.* IEEE Transactions on Information Theory, IT-22(6):644–654, 1976.

[ElG85] Taher ElGamal: *A public key cryptosystem and a signature scheme based on discrete logarithms.* IEEE Transactions on Information Theory, IT-31:469–472, 1985.

[Fel87] Paul Feldman: *A practical scheme for non-interactive verifiable secret sharing.* In *28th Annual Symposium on Foundations of Computer Science*, S. 427 – 437, 1987.

[FGPY97] Y. Frankel, P. Gemmell, P.D.MacKenzie und M. Yung: *Optimal-resilience proactive public-key cryptosystems.* In *Proc. 38th Annual Symposium on Foundations of Computer Science*, S. 384–393, 1997.

[GHJV96] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides: *Entwurfsmuster.* Addison-Wesley, 1996.

[GJKR96] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk und Tal Rabin: *Robust Threshold DSS Signatures.* Lecture Notes in Computer Science, 1070:354–371, 1996.

[MSY00] Shingo Miyazaki, Kouichi Sakurai und Moti Yung: *On Threshold RSA-Signing with no Dealer.* In Song, JooSeok (Herausgeber): *ISISC'99*, S. 197 – 207. Springer Verlag, 2000.

[Oak98] Scott Oaks: *Java Security.* O'Reilly, 1998.

[Ped91] Torben Pedersen: *A threshold cryptosystem without a trusted party.* Advances in Cryptology–EUROCRYPT'91, 547:522–526, 1991.

[Ped92] Torben Pedersen: *Non-interactive and information-theoretically secure verifiable secret sharing.* Advances in Cryptology–CRYPTO'91, 576:129–140, 1992.

[RSA78] R. L. Rivest, A. Shamir und L. M. Adleman: *a method for obtaining digital signatures and public-key cryptosystems.* Communications of the ACM, 21(2):120–126, 1978.

[Sha79] Adi Shamir: *How to Share a Secret.* Communications of the ACM, 22(11):612–613, November 1979.

[Sho00] Victor Shoup: *Practical Threshold Signatures.* In *Theory and Application of Cryptographic Techniques*, S. 207–220, 2000.

[WMB99] Thomas Wu, Michael Malkin und Dan Boneh: *Building Intrusion Tolerant Applications.* In *Proceedings of the 8th USENIX symposium*, S. 79 – 91, 1999.

[YY96] Adam Young und Moti Yung: *The Dark Side of Black-Box Cryptography.* Advances in Cryptology–CRYPTO'96, S. 89–103, 1996.

[YY97] Adam Young und Moti Yung: *Kleptography: Using Cryptography Against Cryptography.* Advances in Cryptology–EUROCRYPT'97, 1233:62–74, 1997.

[ZSvR00] Lidong Zhou, Fred B. Schneider und Robbert van Renesse: *COCA: A secure distributed on-line certification authority.* Technischer Report, Department of Computer Science, Cornell University, 2000.